HOOMD

0.8.0

Generated by Doxygen 1.5.7.1

Mon Dec 22 10:16:19 2008

Contents

1 Main Page

Welcome to the **user** documentation for HOOMD!

1. Installation Guide
2. Quick Start Tutorial
3. Example Scripts
4. Index of Script Commands

If you are looking for the *developer* documentation that used to be here, you can download it for the latest released version of HOOMD from http://www.ameslab.gov/hoomd/download.html . You can also build it from the source code by following the instructions in Compiling HOOMD.

2 Installation Guide

Contents:

2.1 System Requirements for executing HOOMD on a GPU

- **OS:** Linux, Mac OS X, Windows Vista, or Windows XP
  - **Note:** Mac OS X versions older than 10.5.4 are not supported
- **CPU:** Any x86 or x86_64 processor
- **RAM:** At least 1GB is recommended
- **GPU:** Any CUDA capable card: http://www.nvidia.com/object/cuda_home.html
  - **Recommended** (listed here in the rough order of performance expected, see http://www.ameslab.gov/hoomd/benchmarks.html for actual performance numbers):
    * GTX 280 (fastest)
    * Tesla 10 series

    * GTX 260
    * Tesla 800 series
    * 8800 GTX
    * 8800 GTS 512MB
    * 8800 GT
  - **NOTE!** Some board manufacturers sell *Factory Over-clocked* boards. These may work fine in games where a few bit errors just change the color of an onscreen pixel slightly, but GPGPU applications are extremely sensitive to such errors. HOOMD and other CUDA apps are **known to crash** on over-clocked GPUs.

- **Motherboard:** 16x PCIe slot for the graphics card.
  - If you are installing the card in a second slot, double check you motherboard manual first. Many motherboards will decrease the slots to 8x with 2 cards installed. While HOOMD will still run in such a configuration, it is not recommended for performance reasons.

- **Power** supply: Check with the GPU manufacturer for the power supply requirements for your GPU. If you have a store-bought system, its power supply is likely inadequate and will require upgrading.

2.2 Installing HOOMD in Windows

1. Download the HOOMD installer from http://www.ameslab.gov/hoomd
2. Install Python 2.5 using the installer at http://www.python.org/download/
3. Install the corresponding drivers for your GPU. Go to http://www.nvidia.com and select **download drivers**.
   - Note: The HOOMD download page lists compatible driver version(s).
   - Laptop users, see http://forums.nvidia.com/index.php?showtopic=58191
4. Uninstall any previous version of HOOMD before continuing
5. Double click on the HOOMD installer and follow the on screen steps

HOOMD should now show up on your **Start menu** and the .hoomd file type is registered to execute the script when you double click on one. The command **hoomd** can also be run on the **command line** to start the HOOMD python interpreter.

The start menu includes links to benchmark scripts for checking the performance of your setup as well as some live demos that are fun to watch.

In order to visualize the demos, you must have VMD 1.8.6 installed: http://www.ks.uiuc.edu/Research/vmd/

Check out the Quick Start Tutorial to learn how to use HOOMD.

2.3 Installing HOOMD in Linux

1. Before you download HOOMD, check which python version you have by running

   ```
   python -V
   ```

   in a terminal.

2. Download the matching HOOMD package from http://www.ameslab.gov/hoomd

3. Install the NVIDIA CUDA Toolkit using the installer from http://www.nvidia.com/object/cuda_get.html

   - Note: Make sure you install the toolkit version listed on the HOOMD download page.
   - Add /usr/local/cuda/lib to LD_LIBRARY_PATH (can be modified similar to PATH below)

4. Install the corresponding drivers for your GPU. Go to http://www.nvidia.com and select **download drivers**.

   - Note: The drivers must match with the CUDA toolkit you installed. The HOOMD download page lists compatible driver version(s).

5. Extract the HOOMD package using your favorite tool or by running

   ```
   tar -xvjpf hoomd-0.7.1-Linux-x86_64-Python2.5.tar.bz2
   ```

   in a terminal (modify the filename to match the one you downloaded of course).

   - Note: If you extract this package in /opt as root, it will be installed in a system directory for any user on the system to run. If you prefer/need to have your own local copy, just extract to your home directory.

6. Below, replace EXTRACTED_LOCATION with the location you extracted HOOMD to, i.e. /home/joaander/software/hoomd-0.7.1-Linux-x86_64-Python2.5

7. Add EXTRACTED_LOCATION/bin to your $PATH. See http://www.newlinuxuser.com/howto-add-a-directory-to-my-path-statementvariab for instructions. You will probably need to log out and back in for the path setting to take effect.

After installation is complete, run

```
hoomd
```

in any terminal. You should see something like this:

```
Python 2.4.4 (#1, Nov  4 2007, 17:29:36)
[GCC 4.1.2 (Gentoo 4.1.2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Press CTRL-D to exit.

To test your installation further, you can execute benchmark scripts th check the performance of your setup or run some live demos that are fun to watch. You will find them in EXTRACTED_LOCATION/share/hoomd For example, to run the polymer benchmark on the GPU:

```
cd EXTRACTED_LOCATION/share/hoomd/benchmarks/
./polymer_bmark.hoomd --mode=gpu
```

Note: to visualize the live demos, you need to have VMD 1.8.6 installed: http://www.ks.uiuc.edu/Research/vmd/

Check out the Quick Start Tutorial to learn how to use HOOMD.

2.4 Installing HOOMD in Mac OS X

1. Download the HOOMD package from http://www.ameslab.gov/hoomd
2. Install the NVIDIA CUDA Toolkit using the installer from http://www.nvidia.com/object/cuda_get.html
   - **Note:** Make sure you install the toolkit version listed on the HOOMD download page.
   - **Note:** Make sure to select the *Customize* button and ensure that *CUDAKext* is selected.
   - Add /usr/local/cuda/lib to DYLD_LIBRARY_PATH (see instructions for PATH below)
3. Make sure that the Apple **Software Update** tool has installed all updates related to the video drivers.
4. Extract the HOOMD package using your favorite tool or by running

   ```
   tar -xvjpf hoomd-0.8.0-Darwin-i386-Python2.5.tar.bz2
   ```

in a terminal (modify the filename to match the one you downloaded of course).

- Note: If you extract this package in /opt as root, it will be installed in a system directory for any user on the system to run. If you prefer/need to have your own local copy, just extract to your home directory.

5. Below, replace EXTRACTED_LOCATION with the location you extracted HOOMD to, i.e. /home/joaander/software/hoomd-0.8.0-Darwin-i386-Python2.5
6. Add EXTRACTED_LOCATION/bin to your $PATH. See http://www.newlinuxuser.com/howto-add-a-directory-to-my-path-statementvariab for instructions. You will probably need to log out and back in for the path setting to take effect.

After installation is complete, run

```
hoomd
```

in any terminal. You should see something like this:

```
Python 2.5.1 (r251:54863, Apr 15 2008, 22:57:26)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Press CTRL-D to exit.

To test your installation further, you can execute benchmark scripts th check the performance of your setup or run some live demos that are fun to watch. You will find them in EXTRACTED_LOCATION/share/hoomd For example, to run the polymer benchmark on the GPU:

```
cd EXTRACTED_LOCATION/share/hoomd/benchmarks/
./polymer_bmark.hoomd --mode=gpu
```

Note: to visualize the live demos, you need to have VMD 1.8.6 installed: http://www.ks.uiuc.edu/Research/vmd/

Check out the Quick Start Tutorial to learn how to use HOOMD.

3 Quick Start Tutorial

Example script

So you have HOOMD installed. **Now what!?**

Let's start with the classic MD simulation, the Lennard-Jones liquid. Place *N* particles randomly in a box and allow them to interact with the following potential between pairs of particles:

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

To configure HOOMD to perform this simulation, a simple Python script must be written.

```
from hoomd_script import *

# create 100 random particles of name A
init.create_random(N=100, phi_p=0.01, name='A')

# specify Lennard-Jones interactions between particle pairs
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)

# integrate at constant temperature
integrate.nvt(dt=0.005, T=1.2, tau=0.5)

# run 10,000 time steps
run(10e3)
```

If you don't know Python, don't worry. You can learn everything about it that you need to know for HOOMD scripts here. Of course if you did know more Python (which is a full-fledged object oriented programming language: tutorials at http://www.python.org) then you could make use of its capabilities in setting up complicated simulations.

Running the example

For now, copy and paste the above code into a file *test.hoomd*. Assuming you have installed HOOMD, you can run the simulation script from the command line:

```
$ hoomd test.hoomd
```

And you should see output that looks something like this:

```
HOOMD 0.8.0
Compiled: Tue Oct 28 08:33:32 CDT 2008
Copyright, 2008, Ames Laboratory Iowa State University
-----
http://www.ameslab.gov/hoomd/
This code is the implementation of the algorithms discussed in:
   Joshua A. Anderson, Chris D. Lorenz, and Alex Travesset - 'General
   Purpose Molecular Dynamics Fully Implemented on Graphics Processing
   Units', Journal of Computational Physics 227 (2008) 5342-5359
-----
test.hoomd:004  |  init.create_random(N=100, phi_p=0.01, name='A')
```

```
test.hoomd:007  |  lj = pair.lj(r_cut=3.0)
test.hoomd:008  |  lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)
test.hoomd:011  |  integrate.nvt(dt=0.005, T=1.2, tau=0.5)
test.hoomd:014  |  run(10e3)
 starting run **
Time 00:00:00 | Step 10000 / 10000 | TPS 41990.9 | ETA 00:00:00
Average TPS: 41976.2
---------
-- Neighborlist stats:
449 updates / 20 forced updates
n_neigh_min: 0 / n_neigh_max: 10 / n_neigh_avg: 2.23
bins_min: 0 / bins_max: 4 / bins_avg: 1.5625
 run complete **
```

That's it! You've just run your first simulation with HOOMD.

Understanding the output

The first few lines of output are just a header notifying you which version of HOOMD you are running and when it was compiled along with a link to the website and the reference to the paper discussing the algorithms used in HOOMD.

The simulation output starts at:

```
test.hoomd:004  |  init.create_random(N=100, phi_p=0.01, name='A')
```

Each hoomd_script command prints the file and line where it was run along with the entire text of the command. This can be potentially very useful in debugging problems as it will allow you to hone in on the command in the script that is producing the error message.

When a run() command is executed, HOOMD steps the simulation forward that many time steps. While it is doing so, it periodically prints out status lines like the one seen above.

```
Time 00:00:00 | Step 10000 / 10000 | TPS 41990.9 | ETA 00:00:00
```

- **Time** is the total time spent (so far) in the current simulation in HH:MM:SS (totaled over multiple run() commands)
- **Step current / final** prints the *current* time step the simulation is at and the *final* time step of the run()
- **TPS** is the current rate (in **T**ime steps **P**er **S**econd) at which the simulation is progressing.
- **ETA** is the estimated time to completion of the current run() in HH:MM:SS

Since this run was so short, only one line was printed at the end. Modify the script to run a few million time steps and run it to see what happens in longer simulations. Or just see here...

```
Time 00:00:10 | Step 488501 / 10000000 | TPS 48850.1 | ETA 00:03:14
Time 00:00:20 | Step 976312 / 10000000 | TPS 48781.1 | ETA 00:03:04
Time 00:00:30 | Step 1462718 / 10000000 | TPS 48640.5 | ETA 00:02:55
Time 00:00:40 | Step 1950647 / 10000000 | TPS 48792.8 | ETA 00:02:44
Time 00:00:50 | Step 2436905 / 10000000 | TPS 48625.4 | ETA 00:02:35
Time 00:01:00 | Step 2924701 / 10000000 | TPS 48779.5 | ETA 00:02:25
Time 00:01:10 | Step 3410821 / 10000000 | TPS 48612 | ETA 00:02:15
```

The final bit of output at the end of the run prints statistics from various parts of the computation. In this example, only the neighbor list prints statistics.

```
---------
-- Neighborlist stats:
449 updates / 20 forced updates
n_neigh_min: 0 / n_neigh_max: 10 / n_neigh_avg: 2.23
bins_min: 0 / bins_max: 4 / bins_avg: 1.5625
```

Differently configured simulation scripts may print additional information here.

Dissecting the script

1. The first line of every HOOMD job script (except for comment lines starting with #) must be

   ```
   from hoomd_script import *
   ```

   This line takes the python code of hoomd_script, compiles it and loads it in so it can be used. hoomd_script contains the code for commands such as init.create_random() which is why this line must be first.

2. After hoomd_script has been imported, the system must be initialized before any other command can be executed. In this example, we create a 100 random particles named **A**.

   ```
   init.create_random(N=100, phi_p=0.01, name='A')
   ```

   Here is a good point to call attention to one of HOOMD's nifty features. You can name a particle type **anything you want**. If you want to name a particular particle type 'My ridiculously long particle type name', be my guest. HOOMD doesn't care one way or another as it just stores the string you give it. Just be warned that some software packages only handle names up to a certain length, so some information may be lost when they load mol2 (dump.mol2) or xml (dump.xml) files written by HOOMD.

   Documentation for init.create_random

3. The next line specifies the pair force between particle pairs in the simulation. In the example, we create a Lennard-Jones pair force with a cutoff radius of 3.0.

   ```
   lj = pair.lj(r_cut=3.0)
   ```

   This line has the structure `variable=command` which saves the result of the command for later modification (see why on the next line of the script). In HOOMD, any number of forces can be specified, even zero if that is what you need (use a separate line and variable name for each one). All specified forces are added together during the simulation.

   Documentation for pair.lj

4. The parameters of the force must also be specified before the simulation is run() (you will get an error if you forget). The next line sets the parameters *epsilon*, *sigma*, and *alpha* for the Lennard-Jones force between particles of types A and A.

   ```
   lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)
   ```

   The example script only has a single particle type in the simulation, so one line is sufficient. In more complicated simulations with more than one particle type every unique pair must be specified (i.e. 'A'-'A', 'A'-'B', and 'B'-B'). Use one line like that above for each pair.

5. After that, we choose the integrator to move particles forward in time. Unlike with the forces, there is only one integrator (specifying another will overwrite the first). Also, there **must** be one integrator. If you try running the simulation without one HOOMD will print an error message. Here, we create a Nosé-Hoover thermostat and set the timestep *dt* to be 0.005, the temperature target *T* at 1.2 and the parameter *tau* to 0.5.

   ```
   integrate.nvt(dt=0.005, T=1.2, tau=0.5 )
   ```

   After the initialization and before the run() command, the order in which commands are called doesn't matter. The NVT integrator could just as easily been specified before the pair force.

   Documentation for integrate.nvt

6. Finally, the run command actually takes the job settings previously specified and runs the simulation through time.

   ```
   run(10e3)
   ```

   This simple example only runs for 10,000 steps but real simulations might be run for 10's of millions, spending days of computation time in this single command. There is no limit that there be a single run command in a given script. If your simulation needs to turn off a certain force or change integrators and then continue, you can do that. Just execute the commands to make the changes after the first run and before the second.

   Documentation for run()

The anatomy of a hoomd_script command

The init line is as good an example as any.

```
init.create_random(N=100, phi_p=0.01, name='A')
```

Parts of the command

- **init** - The package. Every command is in its own package to keep it organized
- . - Python syntax needed to access a member of the package
- **create_random** - The command name to run
- (- Python required syntax to note the start of an argument list
- **N=100, phi_p=0.01, name='A'** - Arguments (more on these below)
-) - Python required syntax to note the end of an argument list
- <enter> - Python required syntax to execute the command

About the arguments

Multiple arguments of the form `name=value` are separated by commas. Whitespace is ignored so `name = value` works too. The order of arguments doesn't matter (as long as you specify them by name). I.e. all of the following are identical:

```
init.create_random(N=100, name='A', phi_p=0.01)
init.create_random(phi_p = 0.01, N = 100, name = 'A')
init.create_random(phi_p=0.01, name='A', N=100)
```

Check the documentation for a specific command to see what the arguments are and what they mean (i.e. see init.create_random). Here is a copy of the documentation for init.create_random:

```
init.create_random
        (               N,
                        phi_p,
                        name = "A",
                        min_dist = 1.0
        )

Parameters:
        N       Number of particles to create
        phi_p   Packing fraction of particles in the simulation box
        name    Name of the particle type to create
        min_dist        Minimum distance particles will be separated by
```

First of all, notice in the header for the command that some arguments are listed with an = sign, like `name="A"`. This means that argument has a default value associated with it. If you are happy with the default value, you don't need to specify that argument in the list. In our example, we have always been setting `name='A'` anyways so using

```
init.create_random(N=100, phi_p=0.01)
```

is identical.

Those arguments that are listed without the = sign (N, phi_P here) have no default value and **must** be specified. Python will give you an error if you don't.

init.create_random() doesn't have any optional arguments, but some commands do. Optional arguments will be labeled as such and have a default value of `None`. The documentation for any optional arguments will clearly indicate what occurs when you do or do not specify it in the argument list.

Where to go from here

This quick tutorial is only the tip of the iceberg. There are a lot more resources in the documentation.

- Find out how to control what resources HOOMD uses to execute simulations: Command line options
- Page through the list of all commands you can use to define a simulation: Index of Script Commands
- Examine more complicated scripts that illustrate typical usage scenarios: Example Scripts
- Learn how to compile HOOMD so you can modify the code to suit your needs: Compiling HOOMD

4 Example Scripts

Examples:

- Example 1: Using dump files

  The simulation trajectory is written to a file which can be viewed in visualization software such as VMD

- Example 2: Using IMD

  A running simulation is connected to VMD which displays the current system state updated in real-time

- Example 3: Using the polymer generator

  The polymer generator can generate complicated initial conditions for bead-spring polymer systems

- Example 4: Using arbitrary input files

  None of the built-in generators works for you? Write an input file describing the initial condition and load it in.

4.1 Example 1: Using dump files

This simple example is a simple adaptation of the quick start script. It performs a simulation of a Lennard-Jones liquid, dumping snapshots of the system every 100 time steps.

```
from hoomd_script import *

# create 1000 random particles of name A
init.create_random(N=1000, phi_p=0.01, name='A')

# specify Lennard-Jones interactions between particle pairs
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)

# integrate at constant temperature
integrate.nvt(dt=0.005, T=1.2, tau=0.5)

# dump a .mol2 file for the structure information
dump.mol2(filename='example1.mol2')

# dump a .dcd file for the trajectory
dump.dcd(filename='example1.dcd', period=100)

# run 10,000 time steps
run(10e3)
```

Running this quick simulation should result in two output files being generated in the current working directory: `example.mol2` and `example.dcd`. The .mol2 file generated by dump.mol2 contains the particle names and coordinates at time step 0. If there were any bonds specified, they would be included too. VMD or other applications can read in the .mol2 to obtain this information.

`dump.dcd` includes snapshots of the system state (particle position coordinates only) written every 100 time steps. This file can be loaded into visualization software such as VMD and played as a movie or read for analysis purposes.

If you have VMD installed, you can load up the entire simulation trajectory by running

```
vmd example1.mol2 example1.dcd
```

on the command line or by loading these files using VMD's GUI. For the best visualization, open VMD's *Graphical Representation* menu and set the *Drawing Method* to VDW. The default of lines will draw a seemingly random line through the simulation box. This is actually a dummy bond between particles 0 and 1, as VMD refuses to load a MOL2 file without any bonds specified in it.

4.2 Example 2: Using IMD

Here is the same simulation as **Example 1**, this time configured for real-time display in VMD using the IMD interface.

```
from hoomd_script import *

# create 1000 random particles of name A
init.create_random(N=1000, phi_p=0.01, name='A')

# specify Lennard-Jones interactions between particle pairs
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)

# integrate at constant temperature
integrate.nvt(dt=0.005, T=1.2, tau=0.5)

# dump a .mol2 file for the structure information
dump.mol2(filename='example2.mol2')

# setup the IMD server
analyze.imd(port=54321, period=500)

# run a very long time so the simulation can be watched in VMD
run(1e9)
```

Start the simulation running in HOOMD, then load up VMD. Inside VMD, create a new molecule and load the file `example2.mol2` generated at the beginning of the simulation. Then go to the VMD command window and run the commnand

```
imd connect localhost 54321
```

The particles in the display window should begin moving. The display is of the current state of the simulation, updated in **real-time**. Again, the best visualization is obtained by setting the *Drawing Method* to VDW in VMD's *Graphical Representation* menu.

Switch back to the terminal where HOOMD is running and press CTRL-C to kill the simulation. It is set to run for an extremely long time on purpose to allow ample time to launch VMD and issue the `imd` command.

4.3 Example 3: Using the polymer generator

Here is a more complicated script that generates a system of bead-spring polymers which self-assemble into a hex phase when run for a few million time steps. The polymers are A6B7A6 block copolymers in an implicit solvent. The script also shows a few examples of how writing python code in the script can be handy: here the concentration `phi_P` is a parameter and math operations are performed to calculate the length of the box.

For more information on the model in this script, see

"Micellar crystals in solution from molecular dynamics simulations"

J. Chem. Phys. **128**, 184906 (2008); DOI:10.1063/1.2913522

http://link.aip.org/link/?JCPSA6/128/184906/1

Any of the polymer systems in the paper could be easily run just by changing a few parameters in this script.

```
from hoomd_script import *
import math

# parameters
phi_P = 0.25
n_poly = 600
T = 1.2
polymer1 = dict(bond_len=1.2, type=['A']*6 + ['B']*7 + ['A']*6,
                bond="linear", count=n_poly)

# perform some simple math to find the length of the box
N = len(polymer1['type']) * polymer1['count'];
L = math.pow(math.pi * N / (6.0 * phi_P), 1.0/3.0);

# generate the polymer system
init.create_random_polymers(box=hoomd.BoxDim(L), polymers=[polymer1],
                            separation=dict(A=0.35, B=0.35), seed=12);

# force field setup
harmonic = bond.harmonic()
harmonic.set_coeff('polymer', k=330.0, r0=0.84)
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=0.0)
lj.pair_coeff.set('A', 'B', epsilon=1.0, sigma=1.0, alpha=0.0)
lj.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0, alpha=1.0)

# dump every 100,000 steps
dump.mol2(filename="example3.mol2");
dump.dcd(filename="example3.dcd", period=100000);

# integrate NVT for a bunch of time steps
integrate.nvt(dt=0.005, T=T, tau=0.5)
run(2000)

# uncomment the next run() command if you have a few hours to spare
# running this on a GPU the resulting dump files should show the
```

```
# polymers self-assembling into the hex phase
# run(10e9)
```

4.4 Example 4: Using arbitrary input files

Of course, HOOMD is not limited by the built-in random initial condition generators used in the previous example. You can load in an arbitrary initial condition from a formatted xml file. Here is a simple example demonstrating most of the types of data that can be input (see XML File Format for full documentation of this format):

```
<?xml version="1.0" encoding="UTF-8"?>
<hoomd_xml>
<configuration time_step="0">
<box units="sigma"  Lx="10" Ly="10" Lz="10"/>
<!-- Setup the initial condition to place all particles in a line -->
<position units="sigma">
-3 0 0
-2 0 0
-1 0 0
0 0 0
1 0 0
2 0 0
3 0 0
</position>
<!-- Name the first 3 particles A and the rest B -->
<type>
A
A
A
B
B
B
B
</type>
<!-- Bond the particles together into a polymer chain -->
<bond>
polymer 0 1
polymer 1 2
polymer 2 3
polymer 3 4
polymer 4 5
polymer 5 6
</bond>
<!-- Give the particles a little kick with an initial velocity -->
<velocity units="sigma/tau">
1 2 3
3 2 1
1 0 0
0 1 0
0 0 1
-1 -2 -3
-3 -2 -1
</velocity>
</configuration>
```

```
</hoomd_xml>
```

Copy and paste this data to a file *example4.xml*. The initial conditions can be read into a simulation using the command init.read_xml as shown in the example script below.

```
from hoomd_script import *
import math

# read in the file
init.read_xml(filename="example4.xml")

# example4.xml defines a single polymer: use the same force field as in example 3
# force field setup
harmonic = bond.harmonic()
harmonic.set_coeff('polymer', k=330.0, r0=0.84)
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=0.0)
lj.pair_coeff.set('A', 'B', epsilon=1.0, sigma=1.0, alpha=0.0)
lj.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0, alpha=1.0)

# dump every few steps
dump.mol2(filename="example4.mol2");
dump.dcd(filename="example4.dcd", period=10);

# integrate NVT for a bunch of time steps
integrate.nvt(dt=0.005, T=1.2, tau=0.5)
run(2000)
```

5 Index of Script Commands

Click on an individual command in the list to go to its documentation. Click on the package link at the beginning of each list to get a general overview of all commands in that package.

Initialization commands (init):

- init.read_xml - *Reads initial system state from an XML file.*
- init.create_random - *Generates N randomly positioned particles of the same type.*
- init.create_random_polymers - *Generates any number of randomly positioned polymers of configurable types.*

Pair forces (pair):

- pair.lj - *Lennard-Jones pair force.*
- nlist - *Interface for controlling neighbor list parameters.*

Bond forces (bond):

- bond.harmonic - *Harmonic bond forces.*
- bond.fene - *FENE bond forces.*

Wall forces (wall):

- wall.lj - *Lennard-Jones wall force.*

Other forces (force):

- force.constant - *Constant force.*

Integration methods (integrate):

- integrate.nve - *NVE Integration via Velocity-Verlet.*
- integrate.nvt - *NVT Integration via the Nosé-Hoover thermostat.*
- integrate.bdnvt - *NVT integration via Brownian dynamics.*
- integrate.npt - *NPT Integration via the Nosé-Hoover thermostat, Anderson barostat.*

Change particle or system properties during each time step (update):

- update.rescale_temp - *Rescales particle velocities.*
- update.zero_momentum - *Zeroes system momentum.*
- update.sort - *Sorts particles in memory to improve cache coherency.*

Real time analysis of data (analyze):

- analyze.imd - *Sends simulation snapshots to VMD in real-time.*
- analyze.log - *Logs a number of calculated quanties to a file.*
- analyze.msd - *Calculates the mean-squared displacement of groups of particles and logs the values to a file.*

File output for offline analysis (dump):

- dump.xml - *Writes simulation snapshots in the HOOMD XML format.*
- dump.mol2 - *Writes a simulation snapshot in the MOL2 format.*
- dump.dcd - *Writes simulation snapshots in the DCD format.*

Define groups of particles:

- group_all - *Groups all particles.*
- group_tags - *Groups particles by tag.*
- group_type - *Groups particles by type.*

6 XML File Format

6.1 Overview

Both init.read_xml and dump.xml work with the same XML file format for specifying the system of particles. The format requires a minimal amount of meta-information in an easy to understand human-readable format. One of the key advantages of using XML is that it is also easily machine readable and commonly used, so many parsers exist for it.

The basic outline of a HOOMD xml file looks like this

```
<?xml version="1.0" encoding="UTF-8"?>
<hoomd_xml>
<!-- this is a comment, you can put as many of these in the file
        wherever you wish to. -->
<configuration time_step="0">
        <!-- data nodes go here -->
</configuration>
</hoomd_xml>
```

The first line of the file

```
<?xml version="1.0" encoding="UTF-8"?>
```

is just something that **must** be there to identify that this is an XML file.

The second and last lines signify the start and end of the root node `hoomd_xml`. The contents of the root node is between these begin and end markers.

```
<hoomd_xml>
        <!-- contents of root node -->
</hoomd_xml>
```

Inside the root node is the configuration node.

```
<configuration time_step="0">
        <!-- data nodes go here -->
</configuration>
```

`time_step="0"` is an attribute assigned to the configuration node. You can leave if off if you want and the time step will default to 0. It is used as the initial time step in the simulation when read by init.read_xml. In files written by HOOMD, `time_step` will be set to the value of the time step when the system snapshot was taken.

A number of data nodes can be included as part of the configuration and in any order.

- **box** (Simulation box specification)
- **position** (Particle positions)
- **image** (Particle images)
- **velocity** (Particle velocities)
- **type** (Particle types)
- **bond** (Bonds between particles)
- **wall** (Walls in the simulation box)

Detailed documentation for each node is below.

6.2 Simulation box specification

The <box> node defines the dimensions of the simulation box which particles are placed in.

dump.xml **always** writes this node.

init.read_xml **requires** this node be specified.

Example:

```
<box units="sigma"  Lx="5.1" Ly="9.6" Lz="15.8"/>
```

Attributes:

- *units* Currently unused. Potentially used for a future feature in HOOMD supporting different units for length, energy, etc...
- *Lx* Box length in the **x** direction
- *Ly* Box length in the **y** direction
- *Lz* Box length in the **z** direction

6.3 Particle positions

The <position> node sets the position of each particle in the simulation.

dump.xml *optionally* writes this node.

init.read_xml **requires** this node be specified.

Example:

```
<position units="sigma">
-1.45 2.21 1.56
8.76 1.02 5.60
5.67 8.30 4.67
</position>
```

Attributes:

- *units* Currently unused. Potentially used for a future feature in HOOMD supporting different units for length, energy, etc...

In between the begin and end markers <position> and </position> is a series of floating point numbers in plain-text separated by whitespace. These are read in order *x0 y0 z0 x1 y1 z1 x2 y2 z2 ... x(N-1) y(N-1) z(N-1)*. Note that you do not need to specify the number of particles anywhere, just add as many as you want and init.read_xml will count them. The particular form of the whitespace used does not matter (space, tab, newline, etc...), the example above uses spaces between x, y, and z and newlines between particles merely to make it more easily human-readable.

All particles must be in the box: $x >$ -Lx/2.0 and $x <$ Lx/2.0 and similarly for y and z.

6.4 Particle images

The <image> node sets the box image for each particle in the simulation.

dump.xml *optionally* writes this node.

init.read_xml does not require this node.

When used in an input file, the images specified are used as the initial condition for the simulation.

Example:

```
<image>
-1 -5 12
18 2 -10
13 -5 0
</velocity>
```

The format of the node data is the same as for <position> (see Particle positions), except that the values must be integers. If specifying both position and image in an input file, be certain to include the same number of particles in each, or init.read_xml will generate an error.

Image flags are used to track the movement of particles across the periodic boundary conditions. To unwrap the position of a single particle and see its trajectory as if it did not wrap around the boundary, compute

```
x + ix * Lx
```

where `x` is the particle coordinate, `ix` is the image and `Lx` is the box dimension.

6.5 Particle velocities

The <velocity> node sets the velocity of each particle in the simulation.

dump.xml *optionally* writes this node.

init.read_xml does not require this node.

When used in an input file, the velocities specified are used as the initial condition for the simulation.

Example:

```
<velocity units="sigma/tau">
-0.5 -1.2 0.4
0.6 2.0 0.01
-0.4 3.0 0.0
</velocity>
```

Attributes:

- *units* Currently unused. Potentially used for a future feature in HOOMD supporting different units for length, energy, etc...

The format of the node data is the same as for <position> (see Particle positions). If specifying both position and velocity in an input file, be certain to include the same number of particles in each, or init.read_xml will generate an error.

6.6 Particle types

The <type> node sets the type name of each particle in the simulation.

dump.xml *optionally* writes this node.

init.read_xml **requires** this node be specified.

Example:

```
<type>
A
long_type_name
A
</type>
```

The format of the node data is similar to that of <position> (see Particle positions), except that only one type is specified for each particle. A particle type can be **any** string you want that does not include whitespace (as whitespace is used to signify the next particle in the list). Internally, HOOMD assigns no meaning whatsoever to the value of the string you specify so name your particles in ways that are meaningful to you. When performing tasks such as setting the coefficients of a pair force, the type strings given are simply matched up to those specified in the xml file.

If specifying both position and type in an input file, be certain to include the same number of particles in each, or init.read_xml will generate an error.

6.7 Bonds between particles

The <bond> node specifies any number of bonds between particles in the simulation.

dump.xml *optionally* writes this node.

init.read_xml does not require this node

Example:

```
<bond>
polymer 0 1
backbone 1 2
</bond>
```

The above example creates a bond of type *polymer* between particle 0 and 1 and one of type *backbone* between 1 and 2. The general format is *type0 idx_a0 idx_b0 type1 idx_a1 idx_b1 ....* where type is any string without whitespace, idx_a is the index of the first particle in the bond and idx_b is the index of the second particle in the bond. Each field is separated from the next by whitespace (any number of spaces, tabs, or newlines).

Specifying the bonds themselves doesn't result in any additional forces on particles. For that, you must specify a bond force (see bond).

A typical usage would list many bonds of the same type. Coefficients for the associated bond force can be set separately for each bond type by name (for example, see bond.harmonic).

6.8 Walls in the simulation box

The <wall> node specifies any number of walls in the simulation box. Walls have no meaning unless a wall force such as wall.lj is specified in the simulation.

dump.xml *optionally* writes this node.

init.read_xml does not require this node

Example:

```
<wall>
<coord ox="1.0" oy="2.0" oz="3.0" nx="4.0" ny="5.0" nz="6.0"/>
<coord ox="7.0" oy="8.0" oz="9.0" nx="10.0" ny="11.0" nz="-12.0"/>
</wall>
```

Every wall is specified by a plane. The vector (*ox*, *oy*, *oz*) is the origin, a point in the plane and (*nx*, *ny*, *nz*) is the normal.

7 Command line options

Controlling where a simulation is executed

Any simulation in HOOMD can be run on the CPU or GPU. To control which, set the `-mode` option on the script command line. Valid settings are `cpu` and `gpu`.

```
hoomd some_script.hoomd --mode=cpu
```

When `-mode` is set to `gpu` and no other options are specified, HOOMD runs on the first GPU (number 0) in the system. This may not be desirable if another GPU in the system is faster or a simulation is already running on GPU 0 (HOOMD will run correctly with more than one simulation on a GPU as long as there is enough memory, but the performance penalty is severe). You can select which GPU to run on using the `-gpu` command line option.

```
hoomd some_script.hoomd --gpu=1
```

Note that specifying `-gpu` implies `-mode=gpu`. To find out which id is assigned to each GPU in your system, download the CUDA SDK for your system from `http://www.nvidia.com/object/cuda_get.html` and run the `deviceQuery` sample.

Multiple GPUs can be selected to run on by specifying a comma separated list:

```
hoomd some_script.hoomd --gpu=0,1
```

Any number of GPUs can be specified in the list.

If you run a script without any options

```
hoomd some_script.hoomd
```

HOOMD first checks if there are any GPUs in the system. If it finds one or more, it runs on GPU 0. If none are found, it runs on the CPU.

Controlling error checking on the GPU

Detailed error checking is off by default to enable the best performance. If you have trouble that appears to be caused by the failure of a calculation to run on the GPU, you can run with GPU error checking enabled to check for any errors returned by the GPU.

To do this, run the script with the `-gpu_error_checking` command line option:

```
hoomd some_script.hoomd --gpu_error_checking
```

You can always run

```
hoomd some_script.hoomd --help
```

to get a full list of the available command line options some of which may not be listed here.

8 File Conversion Scripts

A number of file conversion scripts for converting between HOOMD's XML format and other formats are provided in the hoomd/bin directory. These scripts should already be in your PATH and executable from the command line.

8.1 HOOMD XML → LAMMPS input

`hoomd2lammps_input.py` converts a HOOMD XML file to the LAMMPS input format. It handles positions, velocities, and bonds (if present in the XML file). To run, execute the script on the command line:

```
hoomd2lammps_input.py input_file.xml output_file.data
```

Here is the output of the above command when run on share/hoomd/demos/micelle.xml:

```
Found 9500  particles
Mapped particle types:
{'A': 1, 'B': 2}
Found 9000 bonds
Mapped bond types:
{'polymer': 1}
```

The script notifies you how many particles, velocities and bonds it reads. Types must be handled in a special way: HOOMD reads types by name, but LAMMPS requires a positive integer from 1 to the number of types. `hoomd2lammps_input.py` scans the input XML and automatically assigns an integer to every type in the order in which they first appear. The output

```
Mapped particle types:
{'A': 1, 'B': 2}
```

is telling you that type A is labeled 1 in the LAMMPS file and type B is labeled 2. The bond type name to integer id conversion is performed in the same manner as the particle types.

8.2 HOOMD XML → LAMMPS dump

`hoomd2lammps_dump.py` converts a series of HOOMD XML files into the LAMMPS dump format. The output file will be formatted as if generated from the LAMMPS command *dump 1 all atom*. The use of scaled coordinates in the output can be disabled on the command line by specifying the option `-noscale`.

Say you have an entire series of dump files from dump.xml in the current working directory. Then

```
hoomd2lammps_dump.py -o dump.lammpstrj *.xml
```

will read them all in and write the LAMMPS dump file *dump.lammpstrj*. HOOMD writes dump files with the time step zero-padded in the file name, so *.xml will list all dump files in time step order. Any number of files can be specified on the command line.

The particle type name to type id handling is the same as in `hoomd2lammps_input.py`: HOOMD XML -> LAMMPS input

8.3 Other formats

If you need to convert to a different format, open one of the existing scripts and see if it can be modified for your needs. The scripts are written in python and the code is easy to read and modify.

9 Compiling HOOMD

Table of contents:

9.1 Software Prerequisites

HOOMD requires a number of prerequisite software packages and libraries to be compiled.

- Python >= 2.3
- boost >= 1.32.0
- CMake >= 2.6.0
  - >= 2.6.1 is recommended on Windows due to an annoying bug in 2.6.0
- Compiler to build source
  - gcc on Linux
  - Visual Studio Express 2005 on Windows XP
  - all code is standard c++ and should work on nearly any other (recent) compiler
  - **note:** Visual Studio 9 (aka 2008) does not currently work because of incompatibilities with boost and CUDA
- CUDA Toolkit and the appropriate NVIDIA display driver:
  - *optional*, but needed to enable GPU support
- Subversion
- Doxygen >= 1.5.6
  - *optional* but needed if you wish to build the detailed developer documentation

Links:

Python - http://www.python.org/

Boost - http://www.boost.org/

CMake - http://www.cmake.org/

CUDA - http://developer.nvidia.com/object/cuda.html

Subversion - http://subversion.tigris.org/

Doxygen - http://www.stack.nl/~dimitri/doxygen/manual.html

9.2 Building on Windows

1. Install prerequisite software

Click for detailed instructions: Installing Software Prerequisites on Windows

2. Get source code

- Option 1) download and unpack source code from: http://www.ameslab.gov/hoomd/
- Option 2) Get the latest development source with subversion This command (assuming you've installed the command line svn tools)

```
$ svn co https://svn2.assembla.com/svn/hoomd/trunk hoomd
```

  will create a directory *hoomd* in your current working directory which will contain the current development version of the source code. You can perform the same operation by right clicking a folder and choosing "SVN Checkout..." when using TortoiseSVN.

Developers with commit access planning to make changes to the code must use option 2.

3. Run CMake

You should now have a directory **hoomd** on your hard drive with a subdirectory **src** containing the source code. CMake must be run to generate the visual studio project that will compile HOOMD.

1. Start cmake-gui
2. Set **C:\Users\joaander\hoomd\src** (modifying to match the location of your hoomd src directory) in the box labeled "Where is the Source code".
3. Set **C:\Users\joaander\hoomd\msvc** (again, modified to mach the location of your hoomd directory) in the box labeled "Where to build the binaries"

4. You should now have a screen that looks like this:
5. Click configure and a dialog pops up:
6. Select the IDE you installed (most likely Visual Studio 8 2005) and click OK.
7. After a short wait while CMake should display a screen that looks like this: If you received an error message instead, it is possible that you are missing one of the prerequisite software packages or it is installed to a non-standard location. In the second case, you can click on the text box with CMAKE-SOMETHING-NOTFOUND in it and specify the full path to the corresponding file or directory.
8. You can configure any of the build options on this screen to your liking. See Build options for more information on what these options do.
9. Click configure several times until all the red options turn white. Then click generate the create project file.
10. Close CMake

4. Compile HOOMD

Open up the **HOOMD.sln** project in visual studio. Press F7 (or use the GUI build button) to build all executables. You can also make a single target the active project (right click and choose set as startup project) and press F7 to build only it.

Source can be modified in visual studio, but any files added to the project must be done via CMake. In most cases, a file can be added simply by placing it in the proper directory and then rerunning CMake.

If you have a system with more than 1 CPU core, you can greatly improve the performance of the build by making use of all cores. In VS2005, navigate to the menu item **Tools->Options**. In the left tab, select **Projects And Solutions->Build and Run**. Set the value for the **maximum number of parallel project builds** to be the total number of CPU cores in your system. There is more information at http://msdn2.microsoft.com/en-us/library/y0xettzf.aspx .

9.3 Building on linux

1. Install prerequisite software

Click for detailed instructions: Installing Software Prerequisites on Linux

2. Get source code

- Option 1) download and unpack source code from: http://www.ameslab.gov/hoomd/
- Option 2) Get the latest development source with subversion This command

```
$ svn co https://svn2.assembla.com/svn/hoomd/trunk hoomd
```

will create a directory *hoomd* in your current working directory which will contain the current development version of the source code.

3. Run CMake

CMake needs to be run to generate the make files to compile HOOMD.

```
$ cd hoomd
$ mkdir bin
$ cd bin
$ cmake-gui ../src
```

Note: If you are logged into the system remotely, you can launch the text mode tool `ccmake` instead of `cmake-gui`. Press "h" for help on using the text mode tool or see http://www.cmake.org/HTML/RunningCMake.html

You will then see a screen that looks like this:

Click configure. You should now see a screen like this: Choose your preferred build environment and click OK. This guide assumes that "Unix Makefiles" was chosen.

Now you should see a screen like this:

If you are building with GPU support enabled, check the ENABLE_CUDA box.

To generate the makefiles now, click **configure** several times until all the red settings turn white, then click **generate**.

You can also scroll down the list of options. Setting the checkbox options will control whether certain features are compiled in. You must click configure after any change. Note that in some cases, changing an option to ON will cause other options to appear. Some options controlled by changing text strings. See Build options for more information on what these options do.

It is possible that some of your libraries may be in non-standard paths. If this is the case, CMake will report an error after you click configure. The offending library will be labeled something like CMAKE_LIB_NOTFOUND. If you know where the library is, you can specify the **full** path here and click configure again.

Make sure to click **generate** again after you make any changes. When you are done, close cmake-gui.

4. Compile HOOMD

CMake generated make files for make. Just run

```
$ make -j4
```

in the **bin** directory to compile everything. This documentation will even be generated if ENABLE_DOXYGEN is selected. The -j4 option lets make compile 4 files at once. It's best to set the value to twice the number of CPU cores in your system.

5(option a). Install to your home directory (so only you can run it)

Re-enter the CMake options screen:

```
$ cmake-gui ../src
```

Set the CMAKE_INSTALL_PREFIX option to a directory of your choice: i.e. /home/joeuser/software/hoomd. Configure, configure and generate again as you did before. When you are back at the command line, execute:

```
$ make install
```

to install HOOMD to the path you specified.

To run HOOMD, either execute /home/joeuser/software/hoomd/bin/hoomd or add /home/joeuser/software/hoomd/bin to your $PATH and run hoomd.

5(option b). Install to a system directory (so any user can run it)

Note: You must be root do do this (or run the "make install" command with sudo). Re-enter the CMake options screen:

```
$ cmake-gui ../src
```

Set the CMAKE_INSTALL_PREFIX option to /opt/hoomd Configure, configure and generate again as you did before. When you are back at the command line, execute:

```
$ make install
```

to install HOOMD to the path you specified.

To make the command `hoomd` available on the $PATH for users, you can either add /opt/hoomd/bin to the system $PATH or make a soft link /usr/bin/hoomd pointing to /opt/hoomd/bin/hoomd.

9.4 Building on Mac OSX

1. Install prerequisite software

Click for detailed instructions: Installing Software Prerequisites on Mac OS X

2. Get source code

- Option 1) download and unpack source code from: http://www.ameslab.gov/hoomd/
- Option 2) Get the latest development source with subversion This command

```
$ svn co https://svn2.assembla.com/svn/hoomd/trunk hoomd
```

will create a directory *hoomd* in your current working directory which will contain the current development version of the source code.

3. Run CMake

First, click on the CMake icon to run CMake. Enter the location where you extracted the HOOMD source code in the "Where is the source code box". Specify where you want the binaries built in the "Where to build the binaries box" (usually in a directory called bin next to the source code). You should now have a screen that looks like this:

Click **configure** and CMake will ask you about the build environment. You have two options here. If there is a particular code editor you prefer and you want to compile by running *make* on the command line, select Unix Makefiles and press enter. If you would rather use XCode as an IDE, select XCode and press enter.

You will then see a screen that looks like this: (this image was captured using the Unix Makefiles generator. Your screen may be slightly different if you chose Xcode)

If you are building with GPU support enabled, check the ENABLE_CUDA box.

To generate the makefiles (or XCode project) now, click **configure** several times until all the red settings turn white, then click **generate**.

You can also scroll down the list of options. Setting the checkbox options will control whether certain features are compiled in. You must click configure after any change. Note that in some cases, changing an option to ON will cause other options to appear. Some options controlled by changing text strings. See Build options for more information on what these options do.

It is possible that some of your libraries may be in non-standard paths. If this is the case, CMake will report an error after you click configure. The offending library will be labeled something like CMAKE_LIB_NOTFOUND. If you know where the library is, you can specify the **full** path here and click configure again.

Make sure to click **generate** again after you make any changes. When you are done, close cmake-gui.

4. Compile HOOMD

If you generated make files, just run

```
$ make -j4
```

in the **bin** directory to compile everything and generate this documentation. The -j4 option lets make compile 4 files at once. It's best to set the value to twice the number of CPU cores in your system.

If you generated an XCode project, open it in XCode and click the build button to compile HOOMD. Note that while source code can be edited via XCode, any files

added to the project must be done with CMake, not the XCode project management. In most cases, this can be done by simply adding the file to the proper directory and rerunning CMake.

5(option a). Install to your home directory (so only you can run it)

Re-enter the CMake options screen:

```
$ cmake-gui ../src
```

Set the CMAKE_INSTALL_PREFIX option to a directory of your choice: i.e. /home/joeuser/software/hoomd. Configure, configure and generate again as you did before. When you are back at the command line, execute:

```
$ make install
```

(or build the INSTALL target in XCode) to install HOOMD to the path you specified.

To run HOOMD, either execute /home/joeuser/software/hoomd/bin/hoomd or add /home/joeuser/software/hoomd/bin to your $PATH and run hoomd.

5(option b). Install to a system directory (so any user can run it)

Note: You must be root do do this (or run the "make install" command with sudo). Re-enter the CMake options screen:

```
$ cmake-gui ../src
```

Set the CMAKE_INSTALL_PREFIX option to /opt/hoomd Configure, configure and generate again as you did before. When you are back at the command line, execute:

```
$ make install
```

(or build the INSTALL target in XCode) to install HOOMD to the path you specified.

To make the command `hoomd` available on the $PATH for users, you can either add /opt/hoomd/bin to the system $PATH or make a soft link /usr/bin/hoomd pointing to /opt/hoomd/bin/hoomd.

9.5 Build options

Here is a list of all the build options that can be changed by CMake.

- **CMAKE_BUILD_TYPE** - sets the build type (Makefile generation only, XCode and Visual Studio can change the build type from within their GUIs)
  - **Debug** - Compiles debug information into the library and executables. Enables asserts to check for programming mistakes. HOOMD will run *very* slow if compiled in Debug mode, but problems are easier to identify.

  - **Release** - All compiler optimizations are enabled and asserts are removed. Recommended for production builds: required for any benchmarking.

- **ENABLE_DOXYGEN** - enables the generation of detailed user and developer documentation
  - Requires doxygen to be installed
- **SINGLE_PRECISION** - Controls precision
  - When set to **ON**, all calculations are performed in single precision.
  - When set to **OFF**, all calculations are performed in double precision.
  - Must be set to **ON** to enable the **ENABLE_CUDA** option (GPUs are single precision)
- **ENABLE_CUDA** - Enable compiling of the GPU accelerated computations using CUDA
  - Requires the CUDA Toolkit to be installed
- **ENABLE_STATIC** - Controls the compiling and linking of static libraries
  - When set to **ON**, **libhoomd** is compiled as a static library and all other libraries (i.e. boost) are linked statically if possible.
  - When set to **OFF**, **libhoomd** is compiled as a dynamic library and all other libraries are linked dynamically if possible.
  - Note: **ENABLE_STATIC=OFF** is not supported on windows.
  - Note 2: **ENABLE_STATIC** defaults ON and can only be set off from the command line and when configuring a clean build directory. Example:

    ```
    ccmake -D ENABLE_STATIC=OFF ../src
    ```

- **ENABLE_VALGRIND** - (Linux only) Runs every unit test through valgrind for hardcore testing/debugging. If used with CUDA, device emulation mode is recommended.

There are a few options for controlling the CUDA compilation.

- **CUDA_BUILD_CUBIN** - Enables a display of register usage for each kernel compiled.
- **CUDA_BUILD_TYPE** - Controls device/emulation builds
  - **Device** - will compile all GPU kernels to run on the GPU hardware
  - **Emulation** - will compile all GPU kernels in a CPU emulation mode. This emulation mode is very slow, but does allow developers without G80 cards to compile and test changes to GPU-related code. Actual kernel development is not recommended without a hardware device to run on.

- **NVCC_USER_FLAGS** - Allows additional flags to be passed to nvcc.
  - If you are building HOOMD for execution solely on G200 and newer GPUs, set NVCC_USER_FLAGS to *-arch;sm_13;-DARCH_SM13* to take advantage of the hardware features present in those GPUs. This boosts HOOMD's overall performance.

9.6 Installing Software Prerequisites on Windows

9.6.1 Visual Studio

First things first, you will need a compiler to build C++ applications. If you don't already have one, that is not a problem. You can download and install a fully featured IDE and compiler, **Visual C++ Express 2005** from here: http://www.microsoft.com/express/2005/ . (note, if you search around you will find that there is a newer express verion, 2008, but it is not currently supported by NVIDIA CUDA, has problems with the current version of boost and thus should not be used).

After installing Visual C++ Express, you **absolutely, positively, most certainly MUST** install Service Pack 1 for it. This can be done using Windows Update or by accessing Help->Update from within Visual Studio.

If you install the express edition, you must also follow the link from http://www.microsoft.com/express/2005/ to download the **Windows Platform SDK** (ignore that the platform sdk says it is for windows 2003 server, it works on XP). If you get an option, the preferred installation path is *C:\Program Files\Microsoft Platform SDK*.

Only a few options need to be activated for the install, namely the Windows Core SDK, and only those components you really need for your architecture as shown in this screenshot (shown for the x86 32-bit arch).

All other installation options can be disabled (not all shown).

After installing the SDK, you need to set some paths. Open Visual Studio and select the **Tools** menu and then select **Options**... Navigate to the section **Projects and Solutions -> VC++ Directories**. As shown in this image: (yours may look slightly different from this one).

Using the upper right drop down box labeled **"Show directories for"** add

C:\Program Files\Microsoft Platform SDK\bin to *Executables*

C:\Program Files\Microsoft Platform SDK\include to *Include* files

and **C:\Program Files\Microsoft Platform SDK\lib** to *Library* files

These are accomplished in the image above with the environment variable, but you do not need to do the same, just type in the full path where you installed the platform SDK

directly or use the file chooser dialog box.

Note: If you intend to build installer packages to be redistributed, make sure that the registry key [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\8.0;InstallDir] is set properly (typically C:\Program Files (x86)\Microsoft Visual Studio 8\Common7\IDE). You need this to get proper installation of the VS redistributable dlls required to run HOOMD on user's machines.

9.6.2 Python

Download and install python binaries from http://www.python.org. If you choose to download the boost binaries (below), you *must* install python version **2.5.1**. If you plan to build boost from source, you can install any version of python you like, as long as it is newer than **2.3**.

9.6.3 Boost

Option 1: Install binaries. Easy, but it only supports version **2.5.x** of python (for boost 1.35.0, this may differ for the other boost versions, depending on which version of python boost-consulting compiled them against).

Download the boost 1.35.0 installer from here http://www.boostpro.com/products/free . When you run it, you will get an option screen that looks like this: Select the options as indicated. You may also be interested in the "Multithread, Debug DLL" and "Multithread, DLL" versions of the libraries, but only select those options if you are an expert HOOMD developer and really know all the implications in linking HOOMD to boost dynamically.

Option 2: Compile from source. Difficult, but enables the use of any supported version of python. These instructions are really only intended for advanced users and developers that will be building statically linked builds for download.

Download the source code from http://www.boost.org. This document assumes you have downloaded boost 1.34.1 (these instructions are known not to work for 1.35 without modifications which will be added eventually). Additionally, you must download and install python before continuing. Instructions are above.

Download the source code for

bzip: http://www.bzip.org/downloads.html

zlib: http://www.zlib.net/

The full instructions for building boost are here: http://www.boost.org/more/getting_started/windows.html. A summary of the commands needed are listed below.

1. Download bjam: http://sourceforge.net/project/showfiles.php?group_-

id=7586&package_id=72941 You will need the file **boost-jam-3.1.16-1-ntx86.zip**

2. Extract everything to a nice place. These instructions assume that **boost**, **zlib** and **bzip2** are all extracted to *c:\libraries* and **bjam.exe** is extracted to *c:\libraries\bjam.exe* .

3. To compile, first open a build environment: Start->Microsoft Platform SDK->Open Build Environment Window->Windows XP 32-bit Build Environment->Set Windows XP 32-bit Build Environment (Retail)

4. In the terminal window that opens, cd to the extracted *boost_1_34_1* directory and run the following command, then wait a *long* time for everything to compile. It is easiest to copy the entire command as one line and paste it (right click and choose paste in the command line window). If you extracted anything to different locations, you will need to change the values in the command line.

```
C:\libraries\bjam.exe --toolset="msvc" -sNO_COMPRESSION=
   -sNO_BZIP2= -sNO_ZLIB= -sBZIP2_SOURCE="c:\libraries\bzip2-1.0.4"
   -sZLIB_SOURCE="c:\libraries\zlib-1.2.3" python=2.5 stage
```

Note: bjam will **NOT** give any errors if any paths are specified incorrectly. Adding –debug-configuration to the end of the command line will provide some information, but not enough to really tell what is going on. The only way to be certain that bzip and zlib are being compiled in is to check at the end for compiled libraries containing **zlib** and **bzip2** in their file names (it is easiest to check after the install step below). Fortunately, if a path is entered incorrectly, you can just rerun this command with the modified path and it will only recompile what is needed so it won't take very long.

Then, run the next command (below). It will install boost header files and libraries into c:\opt (you can change the location by changing the value after –prefix).

```
C:\libraries\bjam.exe --toolset="msvc" -sNO_COMPRESSION= -sNO_BZIP2=
    -sNO_ZLIB= -sBZIP2_SOURCE="c:\libraries\bzip2-1.0.4"
        -sZLIB_SOURCE="c:\libraries\zlib-1.2.3" python=2.5
        --prefix=c:\opt install
```

For CMake to find your newly compiled boost libraries, you must set the environment variable **BOOST_ROOT** to c:\opt (or wherever you installed boost).

9.6.4 CUDA

Download and install the latest toolkit from http://www.nvidia.com/object/cuda_get.html#windows. There is no need to install the SDK. If you have a CUDA capable graphics card, you will also need the driver listed on that page in order to execute HOOMD on the GPU. If you do not have a CUDA capable graphics card, you can still install the toolkit and compile HOOMD in CPU emulation mode. It is quite slow, but developers can use it to test any code they work on that touches GPU classes.

9.6.5 CMake

Download the latest CMake installer from here: http://www.cmake.org/HTML/Download.html

9.6.6 Subversion

TortoiseSVN is a nice graphical interface to subversion that integrates into the windows explorer. Download from here: http://tortoisesvn.net/

If you prefer a command line, you can download one from here: http://www.sliksvn.com/en/download

There is nothing against installing both. One useful reason to do this is to enable HOOMD executables to report the svnversion they were compiled from. This can only be done if the command line tool is installed.

9.7 Installing Software Prerequisites on Linux

This page assumes that you have a standard terminal window open. Commands to run will be indicated as below:

```
 $ echo hello
hello
```

" $ " indicates a shell prompt. As demonstrated above, if you type "echo hello", then you should see the same output obtained above on the next line: "hello"

The process for installing software/libraries differs from linux distribution to distribution. In Gentoo (http://www.gentoo.org/)

```
 $ emerge python
```

would install python. Look at your linux distribution's documentation to find how to install packages on your system (i.e. yum, apt-get, up2date, or another). You may need to "su -" to become root before installing.

9.7.1 Python

First, check if python is already installed

```
 $ python -V
Python 2.4.4
```

Make sure that the version is 2.3 or greater. If you get

```
bash: python: command not found
```

or have a version older than 2.3, you will need to upgrade/install python. Note that you will also need the python development libraries which some distributions might separate into into python-devel or some such. The existence of the python development package can be tested by checking the output of

```
 $ ls /usr/include/python2.X/Python.h
/usr/include/python2.X/Python.h
```

where X is replaced with the major version of python that you have (i.e. For python 2.4.4 above, X would be 4). If this returned

```
ls: cannot access /usr/include/python2.X/Python.h: No such file or directory
```

then you do not have the python development libraries installed.

9.7.2 Boost

First, check if boost is already installed

```
 $ grep BOOST_LIB_VERSION /usr/include/boost/version.hpp
//  BOOST_LIB_VERSION must be defined to be the same as BOOST_VERSION
#define BOOST_LIB_VERSION "1_34_1"
```

Make sure that the version is 1_32_0 or newer. If you get

```
grep: /usr/include/boost/version.hpp: No such file or directory
```

then boost is not installed. You can upgrade/install boost with your distribution's package manager. You may need to install the boost-static package to get the static libraries needed by HOOMD.

If your distribution doesn't have a new enough version of boost, you can build it by hand. Go to http://www.boost.org , download the latest source code and unpack it. cd to the source directory and run

```
 $ mkdir /home/user/software
 $ ./configure --with-python-version=2.5 --prefix=/home/user/software/
```

By default, only shared libraries are built. HOOMD expects static libraries by default, so open up Makefile in your favorite text editor and make the following change (for building boost 1.35 or newer):

```
BJAM_CONFIG=variant=release threading=multi link=shared,static
```

Then you can build and install boost with the following commands.

```
$ make
$ make install
```

Be prepared to wait a while: boost takes a long time to compile.

Before running ccmake or cmake-gui, set the following environment variables to the location where you installed boost:

```
$ export BOOST_ROOT=/home/joaander/software
$ ccmake ../src  # or run cmake-gui
... continue with build instructions ...
```

9.7.3 Compiler

These instructions test for the installation of gcc. Other C++ compilers can be used if you wish, though compilations with CUDA enabled are only supported with gcc.

Test if g++ is installed.

```
$ g++ --version
$ g++ (GCC) 4.1.2 (Gentoo 4.1.2)
```

Any version should do. If you get

```
bash: g++: command not found
```

then you will need to install gcc using your distributions package management system.

9.7.4 CMake

It is not very likely that your linux distribution includes CMake by default, but check anyways.

```
 $ cmake --version
cmake version 2.6-patch 1
```

Make sure the version is 2.6 or later. If you have an old version or get

```
bash: cmake: command not found
```

then you will need to upgrade/install CMake. Try your distributions package manager first. I.e. in Gentoo

```
$ emerge cmake
```

If your distribution does not have a cmake package, then you can install it into your home directory by hand. First, download cmake-2.6.1-Linux-i386.tar.gz from the Downloads section at http://www.cmake.org. Unpack the tarball to any location you prefer: this example assumes you are installing it to the ${HOME}/software

```
$ mkdir ~/software
$ mv cmake-2.6.1-Linux-i386.tar.gz ~/software/
$ cd ~/software
$ tar -xvzf cmake-2.6.1-Linux-i386.tar.gz
```

Then you need to put the bin directory for cmake into your path. If you use bash for a shell you can do this by editing ∼/.bashrc. Look for a line with PATH=... and add the cmake directory to the end separated by a colon. If you can't find the line, create it like so.

```
PATH=$PATH:$HOME/software/cmake-2.6.1-Linux-i386/bin
export PATH
```

Restart your bash shell (or open a new one) and try the version check above to test your installation.

9.7.5 CUDA

Even if you do not have the needed graphics hardware to run, you can still install the CUDA toolkit and run executables in emulation mode. The emulation is slow, but will allow you to develop and test any changes you make that affect any of the ∗GPU classes.

CUDA is quite new and it is not likely that there is a package available through your linux distribution. Go to http://developer.nvidia.com/object/cuda.html#downloads and download the latest CUDA toolkit for you architecture and linux distribution. If your distribution isn't listed, pick one that looks close, it will likely work. To install, simply go to the directory where you downloaded the toolkit and run:

```
$ bash NVIDIA_CUDA_Toolkit_2.0_rhel5_x86_64.run
```

Note, this example lists a specific file: change the command to match the file that you downloaded. The file is a self-unpacking and installing script. Just accept the default location if you have root access to install, or install to ∼/CUDA or anywhere else you please.

Open up ∼/.bashrc in your favorite text editor and add the following line:

```
export LD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
export PATH=$PATH:/usr/local/cuda/bin
```

Change the paths on these lines if you did not install to the default location.

If you have a CUDA capable graphics card, you will also need the proper graphics driver version. See the CUDA webpage linked to above for more information.

If you wish, you can download the CUDA SDK from the same website and compile the example files to test your CUDA installation. The CUDA SDK is not required to compile or run HOOMD, however.

9.7.6 Subversion

Subversion is used for version control. You need to install it if you have commit access and are going to work on active development of HOOMD, or if you just want to download and compile the latest and greatest version.

First, see if you already have subversion installed.

```
 $ svn --version
svn, version 1.4.4 (r25188)
```

If you get

```
-bash: svn: command not found
```

then you will need to install it with your distribution's package manager.

9.8 Installing Software Prerequisites on Mac OS X

Here are detailed instructions on installing the prerequisite software for Mac OS X. They list everything that needs to be done from a clean install of the operating system. If you have already installed something listed here, you can skip it of course.

This page assumes that you have a standard terminal window open in some cases. Commands to run will be indicated as below:

```
 $ echo hello
hello
```

" $ " indicates a shell prompt. As demonstrated above, if you type "echo hello", then you should see the same output obtained above on the next line: "hello"

1. Install XCode

Mac OS X doesn't come with a c++ compiler. You need to download and install XCode from here: http://developer.apple.com/tools/xcode/ Install version 2.5 if you have Mac OS X 10.4, or 3.0 if you have Mac OS X 10.5.

2. Install python

If you are running Mac OS X 10.4, you need to download and install the latest version of python (2.5.1) from http://www.python.org/download/ . Users on 10.5 (Leopard) probably skip this step.

3. Download and compile boost

Building boost is a fairly complicated and time consuming process.

Start by downloading **boost_1_36_0.tar.bz2** from http://www.boost.org/. Extract the tarball and configure boost for building with the following commands.

```
 $ tar -xjf boost_1_36_0.tar.bz2
 $ cd boost_1_36_0
 $ ./configure
lding Boost.Jam with toolset darwin...
tools/jam/src/bin.macosxx86/bjam
-n Detecting Python version...
2.5
-n Detecting Python root...
/System/Library/Frameworks/Python.framework/Versions/2.5
-n Unicode/ICU support for Boost.Regex?...
not found.
Generating Boost.Build configuration in user-config.jam...
Generating Makefile...
```

Users who downloaded and installed python from www.python.org may need to run the configure command with the following argument in order for python to properly be detected.

```
 $ ./configure --with-python=/Library/Frameworks/Python.framework/Versions/Current/bin/python
```

Now, the default settings from configure generate unoptimized boost libraries. We can't have that since HOOMD is all about speed, so open the Makefile

```
 $ open Makefile
```

You need to make the following change (for building boost 1.35 or newer):

```
BJAM_CONFIG=variant=release threading=multi link=shared,static
```

Now, run the command

```
 $ make
```

and wait a *long* time for everything to compile. At the end, you should see a message saying

```
...updated 747 targets...
```

Now, you are ready to install the library (requires administrator privleges).

```
$ sudo make install
```

After typing in your password and a considerably shorter wait, you should see

```
...updated 7079 targets...
```

again. Boost is now installed.

You can delete the boost_1_34_1 directory now if you wish. It might be worth saving for a little while until you have compiled HOOMD and know everything is working so that you won't need to go through all the setup steps again. A common error may be to forget the "sudo" in the last command which will result in boost not being installed and no obvious error message as to why. Just run the command again with the sudo if this is the case.

4. CMake

Download and install the CMake 2.6.1 or newer dmg from http://www.cmake.org/.

5. CUDA

Even if you do not have the needed graphics hardware to run, you can still install the CUDA toolkit and run executables in emulation mode. The emulation is slow, but will allow you to develop and test any changes you make that affect any of the ∗GPU classes.

Go to http://developer.nvidia.com/object/cuda.html#downloads and download the latest CUDA toolkit. Double-click on the downloaded package to install it.

One more step needs to be performed so that applications can find the CUDA libraries. Open up ∼/.bash_profile in your favorite text editor and add the following line:

```
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
export PATH=$PATH:/usr/local/cuda/bin
```

Change the paths on these lines if you did not install to the default location.

If you wish, you can download the CUDA SDK from the same website and compile the example files to test your CUDA installation. The CUDA SDK is not required to compile or run HOOMD, however.

6. Subversion

Subversion is used for version control. You need to install it if you have commit access and are going to work on active development of HOOMD, or if you just want to download and compile the latest and greatest version.

Subversion is included in Mac OS X 10.5 and newer. If you are building HOOMD on an older version, subversion on Mac OS X is most easily installed from the dmg from here: http://downloads.open.collab.net/binaries.html

7. Doxygen

If you want to build this documentation from source, you will need to install Doxygen. There is a dmg available for download at http://www.stack.nl/~dimitri/doxygen/ .

10 License

```
Highly Optimized Object-Oriented Molecular Dynamics (HOOMD) Open
Source Software License
Copyright (c) 2008 Ames Laboratory Iowa State University
All rights reserved.

Redistribution and use of HOOMD, in source and binary forms, with or
without modification, are permitted, provided that the following
conditions are met:

* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names HOOMD's
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

Disclaimer

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND
CONTRIBUTORS ``AS IS''  AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.

IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS  BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

11 Credits

HOOMD Developers:

Joshua Anderson, ISU and Ames Lab - **primary developer**

- Initial code architecture design
- Neighbor list generation
- Lennard-Jones forces
- Harmonic bond forces
- NVE and NVT integration
- hoomd_script design & implementation
- hoomd_xml file format design & implementation
- MOL2 & DCD file format writers
- Random polymer generator
- IMD interface for VMD
- Documentation
- analyze.log design and implementation
- analyze.msd implementation

Alex Travesset, ISU and Ames Lab - **his advisor**

- Electrostatic forces

Rastko Sknepnek, ISU and Ames Lab - additional development

- NPT integration

Carolyn Phillips, University of Michigan - additional development

- FENE bond forces
- Shifted LJ forces
- Testing and debugging HOOMD on Mac OS X systems

Source code

Sockets code from VMD is used for the IMDInterface to VMD (http://www.ks.uiuc.edu/Research/vmd/) - Used under the VMD License

This software includes code developed by the Theoretical and Computational Biophysics Group in the Beckman Institute for Advanced Science and Technology at the University of Illinois at Urbana-Champaign.

XML parsing is performed with XML.c from http://www.applied-mathematics.net/tools/xmlParser.html - Used under the BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the Frank Vanden Berghen nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

FindCUDA.cmake (http://www.sci.utah.edu/~abe/FindCuda.html) has been modified slightly and is used as part of the build system - Used under the MIT License

Libraries

(distributed in a linked binary form if you downloaded a **HOOMD** static build)

boost - Used under the Boost Software License, Version 1.0 (http://www.boost.org/LICENSE_1_0.txt)

zlib - Used under the zlib license (http://www.zlib.net/zlib_-license.html)

bzip2 - Used under the bzip2 license (http://www.bzip.org)

12 Namespace Documentation

12.1 Package hoomd_script

Base module for the user-level scripting API.

Packages

- package analyze

  Commands that analyze the system and provide some output.

- package bond

  Commands that specify bond forces.

- package dump

  Commands that dump particles to files.

- package force

  Other types of forces.

- package globals

  Global variables.

- package init

  Data initialization commands.

- package integrate

  Commands that integrate the equations of motion.

- package pair

  Commands that create forces between pairs of particles.

- package update

  Commands that modify the system state in some way.

- package wall

  Commands that specify wall forces.

Classes

- class group

  Defines a group of particles.

Functions

- def run

  Runs the simulation for a given number of time steps.

- def group_type

  Groups particles by type.

- def group_tags

  Groups particles by tag.

- def group_all

  Groups all particles.

12.1.1 Detailed Description

Base module for the user-level scripting API.

hoomd_script provides a very high level user interface for executing simulations using HOOMD. This python module is designed to be imported into python with "from hoomd_script import ∗"

More details to add later...

12.1.2 Function Documentation

12.1.2.1 def hoomd_script.group_all ()

Groups all particles.

Creates a particle group from all particles in the simulation. The group can then be used by other hoomd_script commands (such as analyze.msd) to specify which particles should be operated on.

Particle groups can be combined in various ways to build up more complicated matches. See group for information and examples.

Examples:

```
all = group_all()
```

12.1.2.2 def hoomd_script.group_tags (*tag_min*, *tag_max* = None)

Groups particles by tag.

Parameters:

tag_min First tag in the range to include (inclusive)

tag_max Last tag in the range to include (inclusive)

The second argument (tag_max) is optional. If it is not specified, then a single particle with tag=tag_min will be added to the group.

Creates a particle group from particles that match the given tag range. The group can then be used by other hoomd_script commands (such as analyze.msd) to specify which particles should be operated on.

Particle groups can be combined in various ways to build up more complicated matches. See group for information and examples.

Examples:

```
half1 = group_tags(0, 999)
half2 = group_tags(1000, 1999)
```

12.1.2.3 def hoomd_script.group_type (*type*)

Groups particles by type.

Parameters:

type Name of the particle type to add to the group

Creates a particle group from particles that match the given type. The group can then be used by other hoomd_script commands (such as analyze.msd) to specify which particles should be operated on.

Particle groups can be combined in various ways to build up more complicated matches. See group for information and examples.

Examples:

```
groupA = group_type('A')
groupB = group_type('B')
```

12.1.2.4 def hoomd_script.run (*tsteps*, *profile* = `False`)

Runs the simulation for a given number of time steps.

Parameters:

tsteps Number of timesteps to advance the simulation by

profile Set to true to enable detailed profiling

Examples:

```
run(1000)
run(10e6)
run(10000, profile=True)
```

Execute the run() command to advance the simulation forward in time. During the run, all previously specified analyzers, dumps, updaters and the integrators are executed every so many time steps as specified by their individual periods.

After run() completes, you may change parameters of the simulation (i.e. temperature) and continue the simulation by executing run() again. Time steps are added cumulatively, so calling run(1000) and then run(2000) would run the simulation up to time step 3000.

run() cannot be executed before the system is initialized. In most cases, it also doesn't make sense to execute run() until after pair forces, bond forces, and an integrator have been created.

When *profile* is *True*, a detailed breakdown of how much time was spent in each portion of the calculation is printed at the end of the run. Collecting this timing information can slow the simulation on the GPU by ~5 percent, so only enable profiling for testing and troubleshooting purposes.

12.2 Package hoomd_script.analyze

Commands that analyze the system and provide some output.

Classes

- class imd

  Sends simulation snapshots to VMD in real-time.

- class log

  Logs a number of calculated quanties to a file.

- class msd

Calculates the mean-squared displacement of groups of particles and logs the values to a file.

12.2.1 Detailed Description

Commands that analyze the system and provide some output.

An analyzer examines the system state in some way every *period* time steps and generates some form of output based on the analysis. Check the documentation for individual analyzers to see what they do.

12.3 Package hoomd_script.bond

Commands that specify bond forces.

Classes

- class harmonic

  Harmonic bond forces.

- class fene

  FENE bond forces.

12.3.1 Detailed Description

Commands that specify bond forces.

Bonds add forces between specified pairs of particles and are typically used to model chemical bonds. Bonds between particles are set when an input XML file is read (init.read_xml) or when an another initializer creates them (like init.create_random_polymers)

By themselves, bonds that have been specified in an input file do nothing. Only when you specify a bond force (i.e. bond.harmonic), are forces actually calculated between the listed particles.

12.4 Package hoomd_script.dump

Commands that dump particles to files.

Classes

- class xml

  Writes simulation snapshots in the HOOMD XML format.

- class mol2

  Writes a simulation snapshot in the MOL2 format.

- class dcd

  Writes simulation snapshots in the DCD format.

12.4.1 Detailed Description

Commands that dump particles to files.

Commands in the dump package write the system state out to a file every *period* time steps. Check the documentation for details on which file format each command writes.

12.5 Package hoomd_script.force

Other types of forces.

Classes

- class constant

  Constant force.

12.5.1 Detailed Description

Other types of forces.

This package contains various forces that don't belong in any of the other categories

12.6 Package hoomd_script.globals

Global variables.

Variables

- particle_data = None;

  Global variable that holds the ParticleData shared by all parts of hoomd_script.

- system = None;

  Global variable that holds the System shared by all parts of hoomd_script.

- list forces = [ ]

  Global variable that tracks the all of the force computes specified in the script so far.

- integrator = None;

  Global variable tracking the last _integrator set.

- neighbor_list = None;

  Global variable tracking the system's neighborlist.

- list loggers = [ ]

  Global variable tracking all the loggers that have been created.

12.6.1 Detailed Description

Global variables.

To present a simple procedural user interface, hoomd_script needs to track many variables globally. These are stored here.

User scripts are not intended to access these variables. However, there may be some special cases where it is needed. Any variable defined here can be accessed in a user script by prepending "globals." to the variable name. For example, to access the global ParticleData, a user script can access `globals.particle_data` .

12.7 Package hoomd_script.init

Data initialization commands.

Functions

- def read_xml

  Reads initial system state from an XML file.

- def create_random

Generates N randomly positioned particles of the same type.

- def create_random_polymers

  Generates any number of randomly positioned polymers of configurable types.

12.7.1 Detailed Description

Data initialization commands.

Commands in the init package initialize the particle system. Initialization via any of the commands here must be done before any other command in hoomd_script can be run.

See also:

Quick Start Tutorial

12.7.2 Function Documentation

12.7.2.1 def hoomd_script.init.create_random (*N*, *phi_p*, *name* = `"A"`, *min_dist* = `0.7`)

Generates N randomly positioned particles of the same type.

Parameters:

N Number of particles to create

phi_p Packing fraction of particles in the simulation box

name Name of the particle type to create

min_dist Minimum distance particles will be separated by

Examples:

```
init.create_random(N=2400, phi_p=0.20)
init.create_random(N=2400, phi_p=0.40, min_dist=0.5)
```

N particles are randomly placed in the simulation box. The dimensions of the created box are such that the packing fraction of particles in the box is *phi_p*. The number density *n* is related to the packing fraction by $n = 6/\pi \cdot \phi_P$ assuming the particles have a radius of 0.5. All particles are created with the same type, given by *name*.

12.7.2.2 def hoomd_script.init.create_random_polymers (*box*, *polymers*, *separation*, *seed* = 1)

Generates any number of randomly positioned polymers of configurable types.

Parameters:

box BoxDim specifying the simulation box to generate the polymers in

polymers Specification for the different polymers to create (see below)

separation Separation radii for different particle types (see below)

seed Random seed to use

Any number of polymers can be generated, of the same or different types, as specified in the argument *polymers*. Parameters for each polymer, include bond length, particle type list, bond list, and count.

The syntax is best shown by example. The below line specifies that 600 block copolymers A6B7A6 with a bond length of 1.2 be generated.

```
polymer1 = dict(bond_len=1.2, type=['A']*6 + ['B']*7 + ['A']*6,
                bond="linear", count=600)
```

Here is an example for a second polymer, specifying just 100 polymers made of 4 B beads bonded in a branched pattern

```
polymer2 = dict(bond_len=1.2, type=['B']*4,
                bond=[(0, 1), (1,2), (1,3), (3,4)] , count=100)
```

The *polymers* argument can be given a list of any number of polymer types specified as above. *count* randomly generated polymers of each type in the list will be generated in the system.

In detail:

- *bond_len* defines the bond length of the generated polymers. This should not necesarily be set to the equilibrium bond length! The generator is dumb and doesn't know that bonded particles can be placed closer together than the separation (see below). Thus *bond_len* must be at a minimum set at twice the value of the largest separation radius. An error will be generated if this is not the case.
- *type* is a python list of strings. Each string names a particle type in the order that they will be created in generating the polymer.
- *bond* can be specified as "linear" in which case the generator connects all particles together with bonds to form a linear chain. *bond* can also be given a list if python tuples (see example above). Each tuple in the form of (a,b) specifies that particle `a` of the polymer be bonded to particle `b`.

separation **must** contain one entry for each particle type specified in *polymers* ('A' and 'B' in the examples above). The value given is the separation radius of each particle of that type. The generated polymer system will have no two overlapping particles.

Examples:

```
init.create_random_polymers(box=hoomd.BoxDim(35),
                            polymers=[polymer1, polymer2],
                            separation=dict(A=0.35, B=0.35));

init.create_random_polymers(box=hoomd.BoxDim(31),
                            polymers=[polymer1],
                            separation=dict(A=0.35, B=0.35), seed=52);

init.create_random_polymers(box=hoomd.BoxDim(18,10,25),
                            polymers=[polymer2],
                            separation=dict(A=0.35, B=0.35), seed=12345);
```

With all other parameters the same, create_random_polymers will always create the same system if *seed* is the same. Set a different *seed* (any integer) to create a different random system with the same parameters. Note that different versions of HOOMD *may* generate different systems even with the same seed due to programming changes.

Note:

1. For relatively dense systems (packing fraction 0.4 and higher) the simple random generation algorithm may fail to find room for all the particles and print an error message. There are two methods to solve this. First, you can lower the separation radii allowing particles to be placed closer together. Then setup integrate.nve with the *limit* option set to a relatively small value. A few thousand time steps should relax the system so that the simulation can be continued without the limit or with a different integrator. For extremely troublesome systems, generate it at a very low density and shrink the box with the command ___ (which isn't written yet) to the desired final size.
2. The polymer generator always generates polymers as if there were linear chains. If you provide a non-linear bond topology, the bonds in the initial configuration will be stretched significantly. This normally doesn't pose a problem for harmonic bonds (bond.harmonic) as the system will simply relax over a few time steps, but can cause the system to blow up with FENE bonds (bond.fene).
3. While the custom bond list allows you to create ring shaped polymers, testing shows that such conformations have trouble relaxing and get stuck in tangled configurations. If you need to generate a configuration of rings, you may need to write your own specialized initial configuration generator that writes HOOMD XML input files (see XML File Format). HOOMD's built-in polymer generator attempts to be as general as possible, but unfortunately cannot work in every possible case.
4. The bond type is named 'polymer' must be used in specifying bond coefficients in command such as bond.harmonic

12.7.2.3 def hoomd_script.init.read_xml (*filename*)

Reads initial system state from an XML file.

Parameters:

filename File to read

Examples:

```
init.read_xml(filename="data.xml")
init.read_xml(filename="directory/data.xml")
```

All particles, bonds, etc... are read from the XML file given, setting the initial condition of the simulation. After this command completes, the system is initialized allowing other commands in hoomd_script to be run. For more details on the file format read by this command, see XML File Format.

12.8 Package hoomd_script.integrate

Commands that integrate the equations of motion.

Classes

- class nvt

  NVT Integration via the Nosé-Hoover thermostat.

- class npt

  NPT Integration via the Nosé-Hoover thermostat, Anderson barostat.

- class nve

  NVE Integration via Velocity-Verlet.

- class bdnvt

  NVT integration via Brownian dynamics.

12.8.1 Detailed Description

Commands that integrate the equations of motion.

Commands beginning with integrate. specify the integrator to use when advancing particles forward in time. By default, no integrator is specified. An integrator can specified anywhere before executing the run() command, which will then use the last

integrator set. If a number of integrators are created in the script, the last one is the only one to take effect. For example:

```
integrate.nvt(dt=0.005, T=1.2, tau=0.5)
integrate.nve(dt=0.005)
run(100)
```

In this example, the nvt integration is ignored as the creation of the nve integrator overwrote it.

However, it is valid to run() a number of time steps with one integrator and then replace it with another before the next run().

Some integrators provide parameters that can be changed between runs. In order to access the integrator to change it, it needs to be saved in a variable. For example:

```
integrator = integrate.nvt(dt=0.005, T=1.2, tau=0.5)
run(100)
integrator.set_params(T=1.0)
run(100)
```

This code snippet runs the first 100 time steps with T=1.2 and the next 100 with T=1.0

12.9 Package hoomd_script.pair

Commands that create forces between pairs of particles.

Classes

- class coeff

  Defines pair coefficients.

- class nlist

  Interface for controlling neighbor list parameters.

- class lj

  Lennard-Jones pair force.

12.9.1 Detailed Description

Commands that create forces between pairs of particles.

Generally, pair forces are short range and are summed over all non-bonded particles within a certain cutoff radius of each particle. Any number of pair forces can be defined

in a single simulation. The net force on each particle due to all types of pair forces is summed.

Pair forces require that parameters be set for each unique type pair. Coefficients are set through the aid of the coeff class. To set this coefficients, specify a pair force and save it in a variable

```
my_force = pair.some_pair_force(arguments...)
```

Then the coefficients can be set using the saved variable.

```
my_force.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=0.0)
my_force.pair_coeff.set('A', 'B', epsilon=1.0, sigma=1.0, alpha=0.0)
my_force.pair_coeff.set('B', 'B', epsilon=1.0, sigma=1.0, alpha=1.0)
```

This example set the parameters *epsilon*, *sigma*, and *alpha* (which are used in pair.lj). Different pair forces require that different coefficients are set. Check the documentation of each to see the definition of the coefficients.

See also:

Quick Start Tutorial

12.10 Package hoomd_script.update

Commands that modify the system state in some way.

Classes

- class sort

  Sorts particles in memory to improve cache coherency.

- class rescale_temp

  Rescales particle velocities.

- class zero_momentum

  Zeroes system momentum.

12.10.1 Detailed Description

Commands that modify the system state in some way.

When an updater is specified, it acts on the particle system each time step to change it in some way. See the documentation of specific updaters to find out what they do.

12.11 Package hoomd_script.wall

Commands that specify wall forces.

Classes

- class lj

  Lennard-Jones wall force.

12.11.1 Detailed Description

Commands that specify wall forces.

Walls can add forces to any particles within a certain distance of the wall. Walls are created when an input XML file is read (read.xml).

By themselves, walls that have been specified in an input file do nothing. Only when you specify a wall force (i.e. wall.lj), are forces actually applied between the wall and the particle.

13 Class Documentation

13.1 bdnvt Class Reference

13.1.1 Detailed Description

NVT integration via Brownian dynamics.

integrate.bdnvt performs constant volume, fixed average temperature simuation based on a NVE simulation with added damping and stochastic heat bath forces.

The total added force $\vec{F}$ is

$$\vec{F} = -\gamma \cdot \vec{v} + \vec{F}_{\mathrm{rand}}$$

where $\vec{v}$ is the particle's velocity and $\vec{F}_{\mathrm{rand}}$ is a random force with magnitude chosen via the fluctuation-dissipation theorem to be consistent with the specified drag (*gamma*) and temperature (*T*).

For poor initial conditions that include overlapping atoms, a limit can be specified to the movement a particle is allowed to make in one time step. After a few thousand time steps with the limit set, the system should be in a safe state to continue with unconstrained integration.

Note:

With an active limit, Newton's third law is effectively **not** obeyed and the system can gain linear momentum. Activate the update.zero_momentum updater during the limited bdnvt run to prevent this.

Public Member Functions

- def __init__

  Specifies the BD NVT integrator.

- def set_params

  Changes parameters of an existing integrator.

- def set_gamma

  Sets gamma parameter for a particle type.

13.1.2 Member Function Documentation

13.1.2.1 def __init__ (*self*, *dt*, *T*, *limit* = `None`, *seed* = `0`)

Specifies the BD NVT integrator.

Parameters:

dt Each time step of the simulation run() will advance the real time of the system forward by *dt*

T Temperature of the simuation *T*

limit (optional) Enforce that no particle moves more than a distance of *limit* in a single time step

seed Random seed to use for the run. Otherwise identical simulations with different seeds set will follow different trajectories.

Examples:

```
integrate.bdnvt(dt=0.005, T=1.0, seed=5)
integrator = integrate.bdnvt(dt=5e-3, T=1.0, seed=100)
integrate.bdnvt(dt=0.005, T=1.0, limit=0.01)
```

13.1.2.2 def set_gamma (*self*, *a*, *gamma*)

Sets gamma parameter for a particle type.

Parameters:

a Particle type

gamma γ for particle type (see below for examples)

set_gamma() sets the coefficient γ for a single particle type, identified by name.

The gamma parameter determines how strongly a particular particle is coupled to the stochastic bath. The higher the gamma, the more strongly coupled: see integrate.bdnvt.

If gamma is not set for any particle type will automatically default to 1.0. It is not an error to specify gammas for particle types that do not exist in the simulation. This can be useful in defining a single simulation script for many different types of particles even when some simulations only include a subset.

Examples:

```
bd.set_gamma('A', gamma=2.0)
```

13.1.2.3 def set_params (*self*, *dt* = `None`, *T* = `None`)

Changes parameters of an existing integrator.

Parameters:

dt New time step (if set)

T New temperature (if set)

To change the parameters of an existing integrator, you must save it in a variable when it is specified, like so:

```
integrator = integrate.bdnvt(dt=0.005, T=1.0)
```

Examples:

```
integrator.set_params(dt=0.007)
integrator.set_params(T=2.0)
```

13.2 coeff Class Reference

13.2.1 Detailed Description

Defines pair coefficients.

All pair forces use coeff to specify the coefficients between different pairs of particles indexed by type. The set of pair coefficients is a symmetric matrix defined over all possible pairs of particle types.

There are two ways to set the coefficients for a particular pair force. The first way is to save the pair force in a variable and call set() directly. To see an example of this, see the documentation for the package pair or the Quick Start Tutorial

The second method is to build the coeff class first and then assign it to the pair force. There are some advantages to this method in that you could specify a complicated set of pair coefficients in a separate python file and import it into your job script.

Example (file *force_field.py*):

```
from hoomd_script import *
my_coeffs = coeff();
my_coeffs.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=0.0)
my_coeffs.set('A', 'B', epsilon=1.0, sigma=1.0, alpha=0.0)
my_coeffs.set('B', 'B', epsilon=1.0, sigma=1.0, alpha=1.0)
```

Example job script:

```
from hoomd_script import *
import force_field

.....
my_force = pair.some_pair_force(arguments...)
my_force.pair_coeff = force_field.my_coeffs
```

Public Member Functions

- def set

  Sets parameters for one type pair.

13.2.2 Member Function Documentation

13.2.2.1 def set (*self*, *a*, *b*, *coeffs*)

Sets parameters for one type pair.

Parameters:

a First particle type in the pair

b Second particle type in the pair

coeffs Named coefficients (see below for examples)

Calling set() results in one or more parameters being set for a single type pair. Particle types are identified by name, and parameters are also added by name. Which parameters you need to specify depends on the pair force you are setting these coefficients for, see the corresponding documentation.

All possible type pairs as defined in the simulation box must be specified before executing run(). You will receive an error if you fail to do so. It is not an error, however, to specify coefficients for particle types that do not exist in the simulation. This can be useful in defining a force field for many different types of particles even when some simulations only include a subset.

There is no need to specify coefficients for both pairs 'A','B' and 'B','A'. Specifying only one is sufficient.

Examples:

```
coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
coeff.set('B', 'B', epsilon=2.0, sigma=1.0)
coeff.set('A', 'B', epsilon=1.5, sigma=1.0)
```

Note:

Single parameters can be updated. If both epsilon and sigma have already been set for a type pair, then executing coeff.set('A', 'B', epsilon=1.1) will update the value of epsilon and leave sigma as it was previously set.

13.3 constant Class Reference

13.3.1 Detailed Description

Constant force.

The command force.constant specifies that a constant force should be added to every particle in the simulation.

Public Member Functions

- def __init__

  Specify the constant force.

- def set_force

  Change the value of the force.

- def disable

  Disables the force.

- def enable

  Enables the force.

13.3.2 Member Function Documentation

13.3.2.1 def __init__ (*self*, *fx*, *fy*, *fz*)

Specify the constant force.

Parameters:

fx x-component of the force

fy y-component of the force

fz z-component of the force

Examples:

```
force.constant(fx=1.0, fy=0.5, fz=0.25)
const = force.constant(fx=0.4, fy=1.0, fz=0.5)
```

13.3.2.2 def disable (*self*) [inherited]

Disables the force.

Examples:

```
force.disable()
```

Executing the disable command will remove the force from the simulation. Any run() command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with enable()

To use this command, you must have saved the force in a variable, as shown in this example:

```
force = pair.some_force()
# ... later in the script
force.disable()
```

13.3.2.3 def enable (*self*) [inherited]

Enables the force.

Examples:

```
force.enable()
```

See disable() for a detailed description.

13.3.2.4 def set_force (*self*, *fx*, *fy*, *fz*)

Change the value of the force.

Parameters:

fx New x-component of the force

fy New y-component of the force

fz New z-component of the force

Using set_force() requires that you saved the created constant force in a variable. i.e.

```
const = force.constant(fx=0.4, fy=1.0, fz=0.5)
```

Example:

```
const.set_force(fx=0.2, fy=0.1, fz=-0.5)
```

13.4 dcd Class Reference

13.4.1 Detailed Description

Writes simulation snapshots in the DCD format.

Every *period* time steps a new simulation snapshot is written to the specified file in the DCD file format. DCD only stores particle positions but is decently space efficient and extremely fast to read and write. VMD can load 100's of MiB of trajectory data in mere seconds.

Use in conjunction with dump.mol2 so that VMD has information on the particle names and bond topology.

Due to constraints of the DCD file format, once you stop writing to a file via disable(), you cannot continue writing to the same file, nor can you change the period of the dump at any time. Either of these tasks can be performed by creating a new dump file with the needed settings.

Public Member Functions

- def __init__

  Initialize the dcd writer.

- def disable

  Disables the analyzer.

13.4.2 Member Function Documentation

13.4.2.1 def __init__ (*self*, *filename*, *period*)

Initialize the dcd writer.

Parameters:

filename File name to write to

period Number of time steps between file dumps

Examples:

```
dump.dcd(filename="trajectory.dcd", period=1000)<br>
dcd = dump.dcd(filename"data/dump.dcd", period=1000)
```

13.4.2.2 def disable (*self*) [inherited]

Disables the analyzer.

Examples:

```
analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any run() command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with enable()

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyzer.some_analyzer()
# ... later in the script
analyzer.disable()
```

13.5 fene Class Reference

13.5.1 Detailed Description

FENE bond forces.

The command bond.fene specifies a fene potential energy between every bonded pair of particles in the simulation.

$$V(r) = -kr_0^2 \ln\left(1 - \left(\frac{r}{r_0}\right)^2\right) + V_{\mathrm{WCA}}(r)$$

where $\vec{r}$ is the vector pointing from one particle to the other in the pair and

$$\begin{aligned} V_{\mathrm{WCA}}(r) = & \quad 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] & r < 2^{\frac{1}{6}}\sigma \\ = & \quad 0 & r \geq 2^{\frac{1}{6}}\sigma \end{aligned}$$

Coefficients k, r_0, ε and σ must be set for each type of bond in the simulation using set_coeff().

Note:

Specifying the bond.fene command when no bonds are defined in the simulation results in an error.

Public Member Functions

- def __init__
  Specify the fene bond force.
- def set_coeff
  Sets the fene bond coefficients for a particular bond type.
- def disable
  Disables the force.
- def enable
  Enables the force.

13.5.2 Member Function Documentation

13.5.2.1 def __init__ (*self*)

Specify the fene bond force.

Example:

```
fene = bond.fene()
```

13.5.2.2 def disable (*self*) [inherited]

Disables the force.

Examples:

```
force.disable()
```

Executing the disable command will remove the force from the simulation. Any run() command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with enable()

To use this command, you must have saved the force in a variable, as shown in this example:

```
force = pair.some_force()
# ... later in the script
force.disable()
```

13.5.2.3 def enable (*self*) [inherited]

Enables the force.

Examples:

```
force.enable()
```

See disable() for a detailed description.

13.5.2.4 def set_coeff (*self*, *bond_type*, *k*, *r0*, *sigma*, *epsilon*)

Sets the fene bond coefficients for a particular bond type.

Parameters:

bond_type Bond type to set coefficients for

k Coefficient k in the force

r0 Coefficient r_0 in the force

sigma Coefficient σ in the force

epsilon Coefficient ϵ in the force

Using set_coeff() requires that the specified bond force has been saved in a variable. i.e.

```
fene = bond.fene()
```

Examples:

```
fene.set_coeff('polymer', k=30.0, r0=1.5, sigma=1.0, epsilon= 2.0)
fene.set_coeff('backbone', k=100.0, r0=1.0, sigma=1.0, epsilon= 2.0)
```

The coefficients for every bond type in the simulation must be set before the run() can be started.

13.6 group Class Reference

13.6.1 Detailed Description

Defines a group of particles.

group should not be created dirctly in hoomd_script code. The following methods can be used to create particle groups.

- group_all()
- group_type()
- group_tags()

The above methods assign a descriptive name based on the criteria chosen. That name can be easily changed if desired:

```
groupA = group_type('A')
groupA.name = "my new group name"
```

Once a group has been created, it can be combined with others to form more complicated groups. To create a new group that contains the intersection of all the particles present in two different groups, use the & operator. Similarly, the | operator creates a new group that is the a union of all particles in two different groups.

Examles:

```
# create a group containing all particles in group A and those with
# tags 100-199
groupA = group_type('A')
group100_199 = group_tags(100, 199);
group_combined = groupA | group100_199;

# create a group containing all particles in group A that also have
# tags 100-199
groupA = group_type('A')
group100_199 = group_tags(100, 199);
group_combined = groupA & group100_199;
```

13.7 harmonic Class Reference

13.7.1 Detailed Description

Harmonic bond forces.

The command bond.harmonic specifies a harmonic potential energy between every bonded pair of particles in the simulation.

$$V(r) = \frac{1}{2}k\left(r - r_0\right)^2$$

where $\vec{r}$ is the vector pointing from one particle to the other in the pair.

Coefficients k and r_0 must be set for each type of bond in the simulation using set_coeff().

Note:

Specifying the bond.harmonic command when no bonds are defined in the simulation results in an error.

Public Member Functions

- def __init__
  Specify the harmonic bond force.
- def set_coeff
  Sets the harmonic bond coefficients for a particular bond type.
- def disable
  Disables the force.
- def enable
  Enables the force.

13.7.2 Member Function Documentation

13.7.2.1 def __init__ (*self*)

Specify the harmonic bond force.

Example:

```
harmonic = bond.harmonic()
```

13.7.2.2 def disable (*self*) [inherited]

Disables the force.

Examples:

```
force.disable()
```

Executing the disable command will remove the force from the simulation. Any run() command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with enable()

To use this command, you must have saved the force in a variable, as shown in this example:

```
force = pair.some_force()
# ... later in the script
force.disable()
```

13.7.2.3 def enable (*self*) [inherited]

Enables the force.

Examples:

```
force.enable()
```

See disable() for a detailed description.

13.7.2.4 def set_coeff (*self*, *bond_type*, *k*, *r0*)

Sets the harmonic bond coefficients for a particular bond type.

Parameters:

bond_type Bond type to set coefficients for

k Coefficient k in the force

r0 Coefficient r_0 in the force

Using set_coeff() requires that the specified bond force has been saved in a variable. i.e.

```
harmonic = bond.harmonic()
```

Examples:

```
harmonic.set_coeff('polymer', k=330.0, r0=0.84)
harmonic.set_coeff('backbone', k=100.0, r0=1.0)
```

The coefficients for every bond type in the simulation must be set before the run() can be started.

13.8 imd Class Reference

13.8.1 Detailed Description

Sends simulation snapshots to VMD in real-time.

analyze.imd listens on a specified TCP/IP port for connections from VMD. Once that connection is established, it begins transmitting simulation snapshots to VMD every *period* time steps.

To connect to a simulation running on the local host, issue the command

```
imd connect localhost 54321
```

in the VMD command window (where 54321 is replaced with the port number you specify for analyze.imd

See also:

Example Scripts

Public Member Functions

- def __init__
  Initialize the IMD interface.
- def disable
  Disables the analyzer.
- def enable
  Enables the analyzer.
- def set_period
  Changes the period between analyzer executions.

13.8.2 Member Function Documentation

13.8.2.1 def __init__ (*self*, *port*, *period*)

Initialize the IMD interface.

Parameters:

port TCP/IP port to listen on

period Number of time steps between file dumps

Examples:

```
analyze.imd(port=54321, period=100)
imd = analyze.imd(port=12345, period=1000)
```

13.8.2.2 def disable (*self*) [inherited]

Disables the analyzer.

Examples:

```
analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any run() command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with enable()

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyzer.some_analyzer()
# ... later in the script
analyzer.disable()
```

13.8.2.3 def enable (*self*) [inherited]

Enables the analyzer.

Examples:

```
analyzer.enable()
```

See disable() for a detailed description.

13.8.2.4 def set_period (*self*, *period*) [inherited]

Changes the period between analyzer executions.

Parameters:

period New period to set

Examples:

```
analyzer.set_period(100);
analyzer.set_period(1);
```

While the simulation is running, the action of each analyzer is executed every *period* time steps.

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyze.some_analyzer()
# ... later in the script
analyzer.set_period(10)
```

13.9 lj Class Reference

13.9.1 Detailed Description

Lennard-Jones pair force.

The command pair.lj specifies that a Lennard-Jones type pair force should be added to every non-bonded particle pair in the simulation.

The force $\vec{F}$ is

$$\begin{aligned} \vec{F} = & -\nabla V(r) & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

where

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \alpha \left(\frac{\sigma}{r}\right)^{6} \right]$$

and $\vec{r}$ is the vector pointing from one particle to the other in the pair.

The following coefficients must be set per unique pair of particle types. See pair or the Quick Start Tutorial for information on how to set coefficients.

- ε - `epsilon`
- σ - `sigma`
- α - `alpha`

Example:

```
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)
```

The cuttoff radius r_{cut} is set once when pair.lj is specified (see __init__())

Public Member Functions

- def __init__

  Specify the Lennard-Jones pair force.

- def disable

  Disables the force.

- def enable

  Enables the force.

13.9.2 Member Function Documentation

13.9.2.1 def __init__ (*self*, *r_cut*)

Specify the Lennard-Jones pair force.

Parameters:

r_cut Cuttoff radius (see documentation above)

Example:

```
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0, alpha=1.0)
```

Note:

Pair coefficients for all type pairs in the simulation must be set before it can be started with run()

13.9.2.2 def disable (*self*) [inherited]

Disables the force.

Examples:

```
force.disable()
```

Executing the disable command will remove the force from the simulation. Any run() command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with enable()

To use this command, you must have saved the force in a variable, as shown in this example:

```
force = pair.some_force()
# ... later in the script
force.disable()
```

13.9.2.3 def enable (*self*) [inherited]

Enables the force.

Examples:

```
force.enable()
```

See disable() for a detailed description.

13.10 lj Class Reference

13.10.1 Detailed Description

Lennard-Jones wall force.

The command wall.lj specifies that a Lennard-Jones type wall force should be added to every particle in the simulation.

The force $\vec{F}$ is

$$\begin{aligned} \vec{F} = & -\nabla V(r) & r < r_{\mathrm{cut}} \\ = & 0 & r \geq r_{\mathrm{cut}} \end{aligned}$$

where

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \alpha \left(\frac{\sigma}{r}\right)^{6} \right]$$

and $\vec{r}$ is the vector pointing from the wall to the particle parallel to the wall's normal.

The following coefficients must be set for each particle type using set_coeff().

- ε - `epsilon`
- σ - `sigma`
- α - `alpha`

Example:

```
lj.set_coeff('A', epsilon=1.0, sigma=1.0, alpha=1.0)
```

The cutoff radius r_{cut} is set once when wall.lj is specified (see __init__())

Public Member Functions

- def __init__
  Specify the Lennard-Jones wall force.
- def set_coeff
  Sets the particle-wall interaction coefficients for a particular particle type.
- def disable
  Disables the force.
- def enable
  Enables the force.

13.10.2 Member Function Documentation

13.10.2.1 def __init__ (*self*, *r_cut*)

Specify the Lennard-Jones wall force.

Parameters:

r_cut Cutoff radius

Example:

```
lj_wall = wall.lj(r_cut=3.0);
```

Note:

Coefficients must be set with set_coeff() before the simulation can be run().

13.10.2.2 def disable (*self*) [inherited]

Disables the force.

Examples:

```
force.disable()
```

Executing the disable command will remove the force from the simulation. Any run() command executed after disabling a force will not calculate or use the force during the simulation. A disabled force can be re-enabled with enable()

To use this command, you must have saved the force in a variable, as shown in this example:

```
force = pair.some_force()
# ... later in the script
force.disable()
```

13.10.2.3 def enable (*self*) [inherited]

Enables the force.

Examples:

```
force.enable()
```

See disable() for a detailed description.

13.10.2.4 def set_coeff (*self*, *particle_type*, *epsilon*, *sigma*, *alpha*)

Sets the particle-wall interaction coefficients for a particular particle type.

Parameters:

particle_type Particle type to set coefficients for

epsilon Coefficient ε in the force

sigma Coefficient σ in the force

alpha Coefficient α in the force

Using set_coeff() requires that the specified wall force has been saved in a variable. i.e.

```
lj_wall = wall.lj(r_cut=3.0)
```

Examples:

```
lj_wall.set_coeff('A', epsilon=1.0, sigma=1.0, alpha=1.0)
lj_wall.set_coeff('B', epsilon=1.0, sigma=2.0, alpha=0.0)
```

The coefficients for every particle type in the simulation must be set before the run() can be started.

13.11 log Class Reference

13.11.1 Detailed Description

Logs a number of calculated quanties to a file.

analyze.log can read a variety of calculated values, like energy and temperature, from specified forces, integrators, and updaters. It writes a single line to the specified output file every *period* time steps. The resulting file is suitable for direct import into a spreadsheet, MATLAB, or other software that can handle simple delimited files.

Quantities that can be logged at any time:

- **num_particles** - Number of particles in the system
- **volume** - Volume of the simulation box
- **temperature** - Temperature of the system
- **pressure** - Pressure of the system
- **kinetic_energy** - Total kinetic energy of the system
- **potential_energy** - Total potential energy of the system
- **conserved_quantity** - Conserved quantity for the current integrator (the actual definition of this value depends on which integrator is being used in the current run()
- **time** - Wall-clock running time from the start of the log in seconds

The following quantities are only available of certain forces have been specified (as noted in the parantheses)

- **pair_lj_energy** (pair.lj) - Total Lennard-Jones potential energy
- **bond_fene_energy** (bond.fene) - Total fene bond potential energy
- **bond_harmonic_energy** (bond.harmonic) - Total harmonic bond potential energy
- **wall_lj_energy** (wall.lj) - Total Lennard-Jones wall energy
- **nvt_xi** (integrate.nvt) - ξ value in the NVT integrator
- **nvt_eta** (integrate.nvt) - η value in the NVT integrator

Public Member Functions

- def __init__

  Initialize the log.

- def set_params

  Change the parameters of the log.

- def disable

  Disables the analyzer.

- def enable

  Enables the analyzer.

- def set_period

  Changes the period between analyzer executions.

13.11.2 Member Function Documentation

13.11.2.1 def __init__ (*self*, *filename*, *quantities*, *period*, *header_prefix* = '')

Initialize the log.

Parameters:

filename File to write the log to

quantities List of quantities to log

period Quantities are logged every *period* time steps

header_prefix (optional) Specify a string to print before the header

Examples:

```
logger = analyze.log(filename='mylog.log', period=100,
                     quantities=['pair_lj_energy'])

analyze.log(quantities=['pair_lj_energy', 'bond_harmonic_energy',
            'kinetic_energy'], period=1000, filename='full.log')

analyze.log(filename='mylog.log', quantities=['pair_lj_energy'],
            period=100, header_prefix='#')

analyze.log(filename='mylog.log', quantities=['bond_harmonic_energy'],
            period=10, header_prefix='Log of harmonic energy, run 5\n')
```

By default, columns in the log file are separated by tabs, suitable for importing as a tab-delimited spreadsheet. The delimiter can be changed to any string using set_params()

The *header_prefix* can be used in a number of ways. It specifies a simple string that will be printed before the header line of the output file. One handy way to use this is to specify header_prefix='#' so that `gnuplot` will ignore the header line automatically. Another use-case would be to specify a descriptive line containing details of the current run. Examples of each of these cases are given above.

13.11.2.2 def disable (*self*) [inherited]

Disables the analyzer.

Examples:

```
analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any run() command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with enable()

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyzer.some_analyzer()
# ... later in the script
analyzer.disable()
```

13.11.2.3 def enable (*self*) [inherited]

Enables the analyzer.

Examples:

```
analyzer.enable()
```

See disable() for a detailed description.

13.11.2.4 def set_params (*self*, *quantities* = None, *delimiter* = None)

Change the parameters of the log.

Parameters:

quantities New list of quantities to log (if specified)

delimiter New delimiter between columns in the output file (if specified)

Using set_params() requires that the specified logger was saved in a variable when created. i.e.

```
logger = analyze.log(quantities=['pair_lj_energy',
                     'bond_harmonic_energy', 'nve_kinetic_energy'],
                     period=1000, filename="'full.log')
```

Examples:

```
logger.set_params(quantities=['bond_harmonic_energy'])
logger.set_params(delimiter=',');
logger.set_params(quantities=['bond_harmonic_energy'], delimiter=',');
```

13.11.2.5 def set_period (*self*, *period*) [inherited]

Changes the period between analyzer executions.

Parameters:

period New period to set

Examples:

```
analyzer.set_period(100);
analyzer.set_period(1);
```

While the simulation is running, the action of each analyzer is executed every *period* time steps.

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyze.some_analyzer()
# ... later in the script
analyzer.set_period(10)
```

13.12 mol2 Class Reference

13.12.1 Detailed Description

Writes a simulation snapshot in the MOL2 format.

At the first time step run() after initializing the dump, the state of the particles at that time step is written to the file in the MOL2 file format. The intended usage is to generate a single structure file that can be used by VMD for reading in particle names and bond topology Use in conjunction with dump.dcd for reading the full simulation trajectory into VMD.

Public Member Functions

- def __init__

  Initialize the mol2 writer.

13.12.2 Member Function Documentation

13.12.2.1 def __init__ (*self*, *filename*)

Initialize the mol2 writer.

Parameters:

filename File name to write to

Examples:

```
dump.mol2(filename="structure.mol2")
```

13.13 msd Class Reference

13.13.1 Detailed Description

Calculates the mean-squared displacement of groups of particles and logs the values to a file.

analyze.msd can be given any number of groups of particles. Every *period* time steps, it calculates the mean squared displacement of each group (referenced to the particle positions at the time step the command is issued at) and prints the calculated values out to a file.

The mean squared displacement (MSD) for each group is calculated as:

$$\langle |\vec{r} - \vec{r}_0|^2 \rangle$$

The file format is the same convient delimited format used by analyze.log

Public Member Functions

- def __init__

  Initialize the msd calculator.

- def set_params

  Change the parameters of the msd analysis.

- def disable

  Disables the analyzer.

- def enable

  Enables the analyzer.

- def set_period

  Changes the period between analyzer executions.

13.13.2 Member Function Documentation

13.13.2.1 def __init__ (*self*, *filename*, *groups*, *period*, *header_prefix* = '')

Initialize the msd calculator.

Parameters:

filename File to write the data to

groups List of groups to calculate the MSDs of

period Quantities are logged every *period* time steps

header_prefix (optional) Specify a string to print before the header

Examples:

```
msd = analyze.msd(filename='msd.log', groups=[group1, group2],
                  period=100)

analyze.log(groups=[group1, group2, group3], period=1000,
            filename='msd.log', header_prefix='#')

analyze.log(filename='msd.log', groups=[group1], period=10,
            header_prefix='Log of group1 msd, run 5\n')
```

A group variable (`groupN` above) can be created by any number of group creation functions. see group for a list.

By default, columns in the file are separated by tabs, suitable for importing as a tab-delimited spreadsheet. The delimiter can be changed to any string using set_params()

The *header_prefix* can be used in a number of ways. It specifies a simple string that will be printed before the header line of the output file. One handy way to use this is to specify header_prefix='#' so that `gnuplot` will ignore the header line automatically. Another use-case would be to specify a descriptive line containing details of the current run. Examples of each of these cases are given above.

13.13.2.2 def disable (*self*) `[inherited]`

Disables the analyzer.

Examples:

```
analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any run() command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with enable()

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyzer.some_analyzer()
# ... later in the script
analyzer.disable()
```

13.13.2.3 def enable (*self*) [inherited]

Enables the analyzer.

Examples:

```
analyzer.enable()
```

See disable() for a detailed description.

13.13.2.4 def set_params (*self*, *delimiter* = None)

Change the parameters of the msd analysis.

Parameters:

delimiter New delimiter between columns in the output file (if specified)

Using set_params() requires that the specified msd was saved in a variable when created. i.e.

```
msd = analyze.msd(filename='msd.log', groups=[group1, group2], period=100)
```

Examples:

```
msd.set_params(delimiter=',');
```

13.13.2.5 def set_period (*self*, *period*) [inherited]

Changes the period between analyzer executions.

Parameters:

period New period to set

Examples:

```
analyzer.set_period(100);
analyzer.set_period(1);
```

While the simulation is running, the action of each analyzer is executed every *period* time steps.

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyze.some_analyzer()
# ... later in the script
analyzer.set_period(10)
```

13.14 nlist Class Reference

13.14.1 Detailed Description

Interface for controlling neighbor list parameters.

A neighbor list should not be directly created by you. One will be automatically created whenever a pair force is specified. The cutoff radius is set to the maximum of that set for all defined pair forces.

Any bonds defined in the simulation are automatically used to exclude bonded particle pairs from appearing in the neighbor list.

Public Member Functions

- def set_params

  Change neighbor list parameters.

13.14.2 Member Function Documentation

13.14.2.1 def set_params (*self*, *r_buff* = `None`, *check_period* = `None`)

Change neighbor list parameters.

Parameters:

r_buff (if set) changes the buffer radius around the cutoff

check_period (if set) changes the period (in time steps) between checks to see if the neighbor list needs updating

set_params() changes one or more parameters of the neighbor list. *r_buff* and *check_-period* can have a significant effect on performance. As *r_buff* is made larger, the neighbor list needs to be updated less often, but more particles are included leading to slower force computations. Smaller values of *r_buff* lead to faster force computation, but more often neighbor list updates, slowing overall performance again. The sweet

spot for the best performance needs to be found by experimentation. The default of *r_buff* = 0.8 works well in practice for Lennard-Jones liquid simulations.

As *r_buff* is changed, *check_period* must be changed correspondingly. The neighbor list is updated no sooner than *check_period* time steps after the last update. If *check_period* is set too high, the neighbor list may not be updated when it needs to be.

For safety, the default check_period is 1 to ensure that the neighbor list is always updated when it needs to be. Increasing this to an appropriate value for your simulation can lead to performance gains of approximately 2 percent.

check_period should be set so that no particle moves a distance more than *r_buff/2.0* during a the *check_period*. If this occurs, a **dangerous build** is counted and printed in the neighbor list statistics at the end of a run().

A single global neighbor list is created for the entire simulation. Change parameters by using the built-in variable **nlist**.

Examples:

```
nlist.set_params(r_buff = 0.9)
nlist.set_params(check_period = 11)
nlist.set_params(r_buff = 0.7, check_period = 4)
```

13.15 npt Class Reference

13.15.1 Detailed Description

NPT Integration via the Nosé-Hoover thermostat, Anderson barostat.

integrate.npt performs constant pressure, constant temperature simulations using the standard Nosé-Hoover thermosta/Anderson barostat.

Public Member Functions

- def __init__

  Specifies the NPT integrator.

- def set_params

  Changes parameters of an existing integrator.

13.15.2 Member Function Documentation

13.15.2.1 def __init__ (*self*, *dt*, *T*, *tau*, *P*, *tauP*)

Specifies the NPT integrator.

Parameters:

dt Each time step of the simulation run() will advance the real time of the system forward by *dt*

T Temperature set point for the Nosé-Hoover thermostat

P Pressure set point for the Anderson barostat

tau Coupling constant for the Nosé-Hoover thermostat.

tauP Coupling constant for the barostat

τ is related to the Nosé mass Q by

$$\tau = \sqrt{\frac{Q}{gk_BT_0}}$$

where g is the number of degrees of freedom, and T_0 is the temperature set point (*T* above).

Examples:

```
integrate.npt(dt=0.005, T=1.0, tau=0.5, tauP=1.0, P=2.0)
integrator = integrate.npt(tau=1.0, dt=5e-3, T=0.65, tauP = 1.2, P=2.0)
```

13.15.2.2 def set_params (*self*, *dt* = None, *T* = None, *tau* = None, *P* = None, *tauP* = None)

Changes parameters of an existing integrator.

Parameters:

dt New time step delta (if set)

T New temperature (if set)

tau New coupling constant (if set)

P New pressure (if set)

tauP New barostat coupling constant (if set)

To change the parameters of an existing integrator, you must save it in a variable when it is specified, like so:

```
integrator = integrate.npt(tau=1.0, dt=5e-3, T=0.65)
```

Examples:

```
integrator.set_params(dt=0.007)
integrator.set_params(tau=0.6)
integrator.set_params(dt=3e-3, T=2.0, P=1.0)
```

13.16 nve Class Reference

13.16.1 Detailed Description

NVE Integration via Velocity-Verlet.

integrate.nve performs constant volume, constant energy simulations using the standard Velocity-Verlet method. For poor initial conditions that include overlapping atoms, a limit can be specified to the movement a particle is allowed to make in one time step. After a few thousand time steps with the limit set, the system should be in a safe state to continue with unconstrained integration.

Note:

With an active limit, Newton's third law is effectively **not** obeyed and the system can gain linear momentum. Activate the update.zero_momentum updater during the limited nve run to prevent this.

Public Member Functions

- def __init__

  Specifies the NVE integrator.

- def set_params

  Changes parameters of an existing integrator.

13.16.2 Member Function Documentation

13.16.2.1 def __init__ (*self*, *dt*, *limit* = `None`)

Specifies the NVE integrator.

Parameters:

dt Each time step of the simulation run() will advance the real time of the system forward by *dt*

limit (optional) Enforce that no particle moves more than a distance of *limit* in a single time step

Examples:

```
integrate.nve(dt=0.005)
integrator = integrate.nve(dt=5e-3)
integrate.nve(dt=0.005, limit=0.01)
```

13.16.2.2 def set_params (*self*, *dt* = None)

Changes parameters of an existing integrator.

Parameters:

dt New time step (if set)

To change the parameters of an existing integrator, you must save it in a variable when it is specified, like so:

```
integrator = integrate.nve(dt=0.005)
```

Examples:

```
integrator.set_params(dt=0.007)
integrator.set_params(dt=3e-3)
```

13.17 nvt Class Reference

13.17.1 Detailed Description

NVT Integration via the Nosé-Hoover thermostat.

integrate.nvt performs constant volume, constant temperature simulations using the standard Nosé-Hoover thermostat.

Public Member Functions

- def __init__
  Specifies the NVT integrator.
- def set_params
  Changes parameters of an existing integrator.

13.17.2 Member Function Documentation

13.17.2.1 def __init__ (*self*, *dt*, *T*, *tau*)

Specifies the NVT integrator.

Parameters:

dt Each time step of the simulation run() will advance the real time of the system forward by *dt*

T Temperature set point for the Nosé-Hoover thermostat

tau Coupling constant for the Nosé-Hoover thermostat.

τ is related to the Nosé mass Q by

$$\tau = \sqrt{\frac{Q}{gk_BT_0}}$$

where g is the number of degrees of freedom, and T_0 is the temperature set point (*T* above).

Examples:

```
integrate.nvt(dt=0.005, T=1.0, tau=0.5)
integrator = integrate.nvt(tau=1.0, dt=5e-3, T=0.65)
```

13.17.2.2 def set_params (*self*, *dt* = None, *T* = None, *tau* = None)

Changes parameters of an existing integrator.

Parameters:

dt New time step delta (if set)

T New temperature (if set)

tau New coupling constant (if set)

To change the parameters of an existing integrator, you must save it in a variable when it is specified, like so:

```
integrator = integrate.nvt(tau=1.0, dt=5e-3, T=0.65)
```

Examples:

```
integrator.set_params(dt=0.007)
integrator.set_params(tau=0.6)
integrator.set_params(dt=3e-3, T=2.0)
```

13.18 rescale_temp Class Reference

13.18.1 Detailed Description

Rescales particle velocities.

Every *period* time steps, particle velocities are rescaled by equal factors so that they are consistent with a given temperature in the equipartition theorem $\langle 1/2mv^2 \rangle = k_BT$.

update.rescale_temp is best coupled with the NVE integrator.

Public Member Functions

- def __init__

  Initialize the rescaler.

- def set_params

  Change rescale_temp parameters.

- def disable

  Disables the updater.

- def enable

  Enables the updater.

- def set_period

  Changes the period between updater executions.

13.18.2 Member Function Documentation

13.18.2.1 def __init__ (*self*, *T*, *period* = 1)

Initialize the rescaler.

Parameters:

T Temperature set point

period Velocities will be rescaled every *period* time steps

Examples:

```
update.rescale_temp(T=1.2)
rescaler = update.rescale_temp(T=0.5)
update.rescale_temp(period=100, T=1.03)
```

13.18.2.2 def disable (*self*) [inherited]

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any run() command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with enable()

To use this command, you must have saved the updater in a variable, as shown in this example:

```
updater = update.some_updater()
# ... later in the script
updater.disable()
```

13.18.2.3 def enable (*self*) [inherited]

Enables the updater.

Examples:

```
updater.enable()
```

See disable() for a detailed description.

13.18.2.4 def set_params (*self*, *T* = None)

Change rescale_temp parameters.

Parameters:

T New temperature set point

To change the parameters of an existing updater, you must have saved it when it was specified.

```
rescaler = update.rescale_temp(T=0.5)
```

Examples:

```
rescaler.set_params(T=2.0)
```

13.18.2.5 def set_period (*self*, *period*) [inherited]

Changes the period between updater executions.

Parameters:

period New period to set

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps.

To use this command, you must have saved the updater in a variable, as shown in this example:

```
updater = update.some_updater()
# ... later in the script
updater.set_period(10)
```

13.19 sort Class Reference

13.19.1 Detailed Description

Sorts particles in memory to improve cache coherency.

Every *period* time steps, particles are reordered in memory based on a Hilbert curve. This operation is very efficient, and the reordered particles significantly improve performance of all other algorithmic steps in HOOMD.

The reordering is accomplished by placing particles in spatial bins *bin_width* distance units wide. A Hilbert curve is generated that traverses these bins and particles are reordered in memory in the same order in which they fall on the curve. Testing indicates that a bin width equal to the particle diameter works well, though it may lead to excessive memory usage in extremely low density systems. set_params() can be used to increase the bin width in such situations.

Because all simulations benefit from this process, a sorter is created by default. If you have reason to disable it or modify parameters, you can use the built-in variable `sorter` to do so after initialization. The following code example disables the sorter. The init.create_random command is just an example, sorter can be modified after any command that initializes the system.

```
init.create_random(N=1000, phi_p=0.2)
sorter.disable()
```

Public Member Functions

- def __init__

  Initialize the sorter.

- def set_params

Change sorter parameters.

- def disable

  Disables the updater.

- def enable

  Enables the updater.

- def set_period

  Changes the period between updater executions.

13.19.2 Member Function Documentation

13.19.2.1 def __init__ (*self*)

Initialize the sorter.

Users should not initialize the sorter directly. One in created for you when any initialization command from init is run. The created sorter can be accessed via the built-in variable `sorter`.

By default, the sorter is created with a *bin_width* of 1.0 and an update period of 500 time steps. The period can be changed with set_period() and the bin width can be changed with set_params()

13.19.2.2 def disable (*self*) `[inherited]`

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any run() command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with enable()

To use this command, you must have saved the updater in a variable, as shown in this example:

```
updater = update.some_updater()
# ... later in the script
updater.disable()
```

13.19.2.3 def enable (*self*) [inherited]

Enables the updater.

Examples:

```
updater.enable()
```

See disable() for a detailed description.

13.19.2.4 def set_params (*self*, *bin_width* = None)

Change sorter parameters.

Parameters:

bin_width New bin width (if set)

Examples:

```
sorter.set_params(bin_width=2.0)
```

13.19.2.5 def set_period (*self*, *period*) [inherited]

Changes the period between updater executions.

Parameters:

period New period to set

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps.

To use this command, you must have saved the updater in a variable, as shown in this example:

```
updater = update.some_updater()
# ... later in the script
updater.set_period(10)
```

13.20 xml Class Reference

13.20.1 Detailed Description

Writes simulation snapshots in the HOOMD XML format.

Every *period* time steps, a new file will be created. The state of the particles at that time step is written to the file in the HOOMD XML format.

See also:

XML File Format

Public Member Functions

- def __init__
  Initialize the hoomd_xml writer.
- def set_params
  Change xml write parameters.
- def disable
  Disables the analyzer.
- def enable
  Enables the analyzer.
- def set_period
  Changes the period between analyzer executions.

13.20.2 Member Function Documentation

13.20.2.1 def __init__ (*self*, *filename*, *period*)

Initialize the hoomd_xml writer.

Parameters:

filename Base of the time name

period Number of time steps between file dumps

Examples:

```
dump.xml(filename="atoms.dump", period=1000)
xml = dump.xml(filename="particles", period=1e5)
```

A new file will be created every *period* steps. The time step at which the file is created is added to the file name in a fixed width format to allow files to easily be read in order. I.e. the write at time step 0 with `filename="particles"` produces the file `particles.0000000000.xml`

By default, only particle positions are output to the dump files. This can be changed with set_params().

13.20.2.2 def disable (*self*) [inherited]

Disables the analyzer.

Examples:

```
analyzer.disable()
```

Executing the disable command will remove the analyzer from the system. Any run() command executed after disabling an analyzer will not use that analyzer during the simulation. A disabled analyzer can be re-enabled with enable()

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyzer.some_analyzer()
# ... later in the script
analyzer.disable()
```

13.20.2.3 def enable (*self*) [inherited]

Enables the analyzer.

Examples:

```
analyzer.enable()
```

See disable() for a detailed description.

13.20.2.4 def set_params (*self*, *position* = None, *image* = None, *velocity* = None, *type* = None, *wall* = None, *bond* = None)

Change xml write parameters.

Parameters:

position (if set) Set to True/False to enable/disable the output of particle positions in the xml file

image (if set) Set to True/False to enable/disable the output of particle images in the xml file

velocity (if set) Set to True/False to enable/disable the output of particle velocities in the xml file

type (if set) Set to True/False to enable/disable the output of particle types in the xml file

wall (if set) Set to True/False to enable/disable the output of walls in the xml file

bond (if set) Set to True/False to enable/disable the output of bonds in the xml file

Using set_params() requires that the dump was saved in a variable when it was specified.

```
xml = dump.xml(filename="particles", period=1e5)
```

Examples:

```
xml.set_params(type=False)
xml.set_params(position=False, type=False, velocity=True)
xml.set_params(type=True, position=True)
xml.set_params(position=True, wall=True)
xml.set_params(bond=True)
```

13.20.2.5 def set_period (*self*, *period*) [inherited]

Changes the period between analyzer executions.

Parameters:

period New period to set

Examples:

```
analyzer.set_period(100);
analyzer.set_period(1);
```

While the simulation is running, the action of each analyzer is executed every *period* time steps.

To use this command, you must have saved the analyzer in a variable, as shown in this example:

```
analyzer = analyze.some_analyzer()
# ... later in the script
analyzer.set_period(10)
```

13.21 zero_momentum Class Reference

13.21.1 Detailed Description

Zeroes system momentum.

Every *period* time steps, particle velocities are modified such that the total linear momentum of the system is set to zero.

update.zero_momentum is intended to be used when the NVE integrator has the *limit* option specified, where Newton's third law is broken and systems could gain momentum. However, nothing prevents update.zero_momentum from being used in any HOOMD script.

Public Member Functions

- def __init__

  Initialize the momentum zeroer.

- def disable

  Disables the updater.

- def enable

  Enables the updater.

- def set_period

  Changes the period between updater executions.

13.21.2 Member Function Documentation

13.21.2.1 def __init__ (*self*, *period* = `1`)

Initialize the momentum zeroer.

Parameters:

period Momentum will be zeroed every *period* time steps

Examples:

```
update.zero_momentum()
zeroer= update.zero_momentum(period=10)
```

13.21.2.2 def disable (*self*) [inherited]

Disables the updater.

Examples:

```
updater.disable()
```

Executing the disable command will remove the updater from the system. Any run() command executed after disabling an updater will not use that updater during the simulation. A disabled updater can be re-enabled with enable()

To use this command, you must have saved the updater in a variable, as shown in this example:

```
updater = update.some_updater()
# ... later in the script
updater.disable()
```

13.21.2.3 def enable (*self*) [inherited]

Enables the updater.

Examples:

```
updater.enable()
```

See disable() for a detailed description.

13.21.2.4 def set_period (*self*, *period*) [inherited]

Changes the period between updater executions.

Parameters:

period New period to set

Examples:

```
updater.set_period(100);
updater.set_period(1);
```

While the simulation is running, the action of each updater is executed every *period* time steps.

To use this command, you must have saved the updater in a variable, as shown in this example:

```
updater = update.some_updater()
# ... later in the script
updater.set_period(10)
```

Index